A Mechanically-Checked Correctness Proof of a Floating-Point Search Program

Matt Wilding

Technical Report #56 May, 1990

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151 and by ONR Contract N00014-88-C-0454. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

1. Introduction

This technical report describes work toward the mechanical certification of floating-point program correctness theorem proofs. A library of useful facts about rational numbers was constructed. A set of axioms about floating-point operations has been proved consistent using a mechanical theorem prover that employs the rational number library. A small floating point program has been developed and mechanically-checked correctness theorems about it constructed.

Floating-point numbers are rational numbers that can be expressed in the form

$$sign \times value \times b^{exp}$$

where *sign* is 1 or -1, *value* is a base b number with a number of digits fixed by the floating-point system, and *exp* is in a range fixed by the floating-point system.

Advantages inherent in floating-point numbers were recognized early in the development of modern computers, and many early machines had floating-point capability. [13] Floating-point numbers seemed to make programming easier as operations on very small numbers and very large numbers had the same precision and could usually be handled with little concern about violating range restrictions.

Floating-point arithmetic operations are called *inexact* because they sometimes return values that are "close to" the exact result. An arithmetic operation applied to two rational values that are floating point values may yield a rational value that is not a floating-point value. This inexactness can make analyzing floating-point programs very difficult.

The importance of developing program correctness arguments has become more obvious as programming has matured and programs have become more complex and relied-upon. Techniques for program proof have been developed and applied in many problem domains. There has been much less work has been done to prove the correctness of floating-point programs. Testing practices developed with an understanding of the implementation of floating-point programs are used by some programmers to give them increased confidence in their programs. [15]

Knuth included analysis of floating-point programs in his programming recipes book. [13] One of the techniques for analyzing floating-point arithmetic to which he refers is Wilkinson's "backward" error analysis. [18] Rather than bound the inexactness of floating-point operations by bounding the result, Wilkinson shows that it is often simpler to view the inexact result of floating-point operations as the exact result of perturbed arguments.

Several researchers have also proposed models of floating-point operations and used them to justify claims about sequences of floating point operations. [6, 8, 14, 17] These efforts work toward putting floating-point arithmetic on more-solid ground. Even so, the complexity of proofs using these systems, and the apparent gap between the floating-point axioms and "realistic" numerical programs has discouraged application of work in this area.

Much of the effort toward precisely specifying floating-point operations has been motivated by the desire to implement floating-point operations correctly. Researchers at Oxford University have formalized a floating-point system expressed using Z and Occam [1]. Their ultimate goal is to construct a floating-point processor that correctly implements the formal description of the floating-point system.

The problem of wedding a model of floating-point operations with program proof rules was examined in Holm's PhD thesis. [10] Holm uses an axiomatization of floating-point operations and Dijkstra's WP calculus. [9] Holm develops enough mathematical machinery to prove some theorems about a searching program. While the searching program example may appear unambitious and though the presentation is quite clean, the proof is long and rather complex.

On another front, proofs about computers and computer programs have been mechanically checked. Boyer's and Moore's NQTHM prover has been used to prove the correctness of a microprocessor, compilers, and operating systems. [3, 4] The checked theorems often have convoluted proofs, but they have the important advantage that NQTHM-checked theorems are presumed to be very reliable.

Theorems about floating-point programs appear to be a good domain for mechanical-checking as theorems about floating-point programs are often not realizable by programmers because of the complexity of their proof. Knuth observes that "many a serious mathematician has attempted to give rigorous analyses of a sequence of floating-point operations, but has found the task to be so formidable he has tried to content himself with plausibility arguments instead" [13]. The many details of a proof about floating-point programs are often not of great interest, so the disadvantage of many machine-checked proofs - that the proof is obscure - is usually not relevant in this domain.

Nevertheless, to the author's knowledge no mechanically-checked proof about a floating-point program has ever before been constructed.

PC-NQTHM, the Kaufmann Proof-Checker extension of the NQTHM prover, allows the user finer control of the direction of the prover. [12] It is particularly valuable when the underlying theory is immature, as is currently the case with the current development of rational number arithmetic and floating-point operations. Many of the theorems in this project were proved correct using the tools provided by this interactive enhancement.

This technical report describes a proof of a floating-point search program. Section I is this introduction. Section II is a description of the rational number library that is used to accomplish PC-NQTHM proofs about rationals. Section III describes the axiomatization of a floating-point system and the proof of its consistency. Section IV details the development of the search program, and section V describes its correctness proof. Section VI describes work planned for the future.

The appendices list theorems proved in the project. Appendix A presents an example rational number theorem proof. Appendix B contains the rational library. Appendix C contains an axiomatization of an FP system. Appendix D lists the development and proof of the FP searching example.

2. The Rational Library

In this section we describe a rational number library that contains facts that facilitate automatic proofs about rationals using the PC-NQTHM theorem prover. It uses Bevier's hardware libraries [2] (as updated by Bevier and Wilding) that facilitate proofs about integers and natural numbers.

The floating-point numbers are a subset of the rational numbers, and our later development of them will rely on the definition of rational numbers and the rational number operations contained in the subsection DEFINITIONS. The rest of this section describes the theorems proved about rationals that appear to facilitate mechanical proofs about rationals. Though these rational facts are necessary to the construction of mechanical proofs about floating-point numbers, only the definitions are needed to understand what has been proved.

The definition of operations in rational arithmetic is usually well below the level of detail with which mathematicians work. We develop rational arithmetic formally so that we can apply the general purpose PC-NQTHM prover to our problem. The most natural notation for this section is therefore the Lisp-like syntax of the logic of the prover. In later sections a non-Lisp notation will be used when the proof is not so inextricably intertwined with the operation of the prover.

2.1 Definitions

The rational data type is added using the NQTHM add-shell event.

```
(add-shell rational nil rational-formp
    ((numerator   (one-of numberp negativep) zero)
     (denominator (one-of numberp) zero)))
```

RATIONALP identifies "proper" rational numbers that are of the correct data type, have integer numerators, and non-zero natural number denominators.

```
(definition rationalp (x)
   (and (rational-formp x)
        (integerp (numerator x))
        (not (zerop (denominator x)))))
```

REDUCE reduces a rational to its least common form.

```
(definition reduce (r)
  (if (rationalp r)
    (if (negativep (numerator r))
      (rational (minus (quotient (negative-guts (numerator r))
                       (gcd (negative-guts (numerator r))
                            (denominator r))))
                (quotient (denominator r)
                          (gcd (negative-guts (numerator r))
                               (denominator r))))
      (rational (quotient (numerator r)
                          (gcd (numerator r) (denominator r)))
                (quotient (denominator r)
                          (gcd (numerator r) (denominator r)))))
     (rational 0 1)))
```

FIX-RATIONAL maps non-rationalps to rational 0

```
(definition fix-rational (x)
  (if (rationalp x) x (rational 0 1)))
```

RZEROP identifies non-rationals and rational 0

```
(definition rzerop (x)
  (or (not (rationalp x))
      (equal (numerator x) 0)))
```

RLESSP is the less-than predicate for rationals.

```
(defn rlessp (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (ilessp (itimes (numerator a) (denominator b))
            (itimes (numerator b) (denominator a)))))
```

REQUAL is the equality predicate for rationals.

```
(defn requal (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
     (equal (itimes (numerator a) (denominator b))
            (itimes (numerator b) (denominator a)))))
```

We next define several mathematical operations. Each is defined to return its result in lowest common terms.

```
(definition simple-rplus (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (rational (iplus (itimes (numerator a) (denominator b))
                     (itimes (numerator b) (denominator a)))
              (itimes (denominator a) (denominator b)))))

(definition rplus (x y)
  (reduce (simple-rplus x y)))
```

```
(definition simple-rneg (x)
  (let ((a (fix-rational x)))
    (rational (ineg (numerator a))
              (denominator a))))

(definition rneg (x)
  (reduce (simple-rneg x)))

(definition rdifference (a b)
  (rplus a (rneg b)))

(definition simple-rtimes (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (rational (itimes (numerator a) (numerator b))
              (times (denominator a) (denominator b)))))

(definition rtimes (x y)
  (reduce (simple-rtimes x y)))

(defn simple-rinverse (r)
  (if (rzerop r)
    (rational 0 1)
   (if (negativep (numerator r))
     (rational (ineg (denominator r))
               (ineg (numerator r)))
    (rational (denominator r) (numerator r)))))

(defn rinverse (r)
  (reduce (simple-rinverse r)))

(definition rquotient (x y)
  (rtimes x (rinverse y)))

(definition simple-rmagnitude (x)
  (let ((a (fix-rational x)))
    (if (negativep (numerator a))
      (rneg a)
     a)))

(definition rmagnitude (x)
  (reduce (simple-rmagnitude x)))
```

2.2 Rules

A set of useful rules has been developed to prove things about rational numbers. This section lists the rules in the current rational number library called R2. A sample is given of the application of each rule.

The intention of this subsection is to convey some of the strategy of the rationals library. To understand exactly when and how the rules are applied, one needs to examine the actual rules that are given in Appendix B. Each of the rules in this theory has been proved to be truth-preserving by a NQTHM prove-lemma event. Each rule name is followed by an example of the application of the rule to a NQTHM term. "x --> y" signifies that term x is rewritten to term y if the rule is applied.

```
rtimes-rneg-arg2
   (rtimes x (rneg y)) --> (rneg (rtimes x y))
```

```
rtimes-rneg-arg1
   (rtimes (rneg x) y) --> (rneg (rtimes x y))

rtimes-rplus-arg2
   (rtimes x (rplus y z)) --> (rplus (rtimes x y) (rtimes x z))

rtimes-rplus-arg1
   (rtimes (rplus x y) z) --> (rplus (rtimes x z) (rtimes y z))

associativity-of-rtimes
   (rtimes (rtimes x y) z) --> (rtimes x (rtimes y z))

rzerop
   (rzerop x) -->  (or (not (rationalp x)) (equal (numerator x) 0))

commutativity-of-rtimes
   (rtimes x y) --> (rtimes y x)

rdifference-rdifference-arg2
   (rdifference (rdifference x y) z) --> (rdifference x (rplus y z))

rdifference-rdifference-arg1
   (rdifference x (rdifference y z)) --> (rdifference (rplus x z) y)

rneg-rplus
   (rneg (rplus x y)) --> (rplus (rneg x) (rneg y))

rplus-reduce-arg2-rewrite
   (rplus x (reduce y)) --> (rplus x y)

 rplus-reduce-arg1-rewrite
   (rplus (reduce x) y) --> (rplus x y)

reduce-rneg
   (reduce (rneg x)) --> (rneg x)

reduce-rmagnitude
   (reduce (rmagnitude x)) --> (rmagnitude x)

reduce-rquotient
   (reduce (rquotient x)) --> (rquotient x)

reduce-difference
   (reduce (rdifference x y)) --> (rdifference x y)

reduce-rtimes
   (reduce (rtimes x y)) --> (rtimes x y)

reduce-rplus
   (reduce (rplus x y)) --> (rplus x y)

commutativity2-of-rplus
   (rplus x (rplus y z)) --> (rplus y (rplus x z))

associativity-of-rplus
   (rplus (rplus x y) z) --> (rplus x (rplus y z))

equal-requal-rewrite
   (requal a b) --> (requal a c)     given: (requal b c)

transitivity-of-requal
   (requal a c) --> t                given: (requal a b) and (requal b c)

fix-rational-of-rationalp
   (fix-rational x) --> x            given: (rationalp x)

nrational-rplus-arg2
   (rplus y x) --> (reduce y)        given: (not (rationalp x))
```

```
nrational-rplus-arg1
   (rplus x y) --> (reduce y)
```
given: (not (rationalp x))

```
rationalp-means
   (rational-formp x) --> t
   (integerp (numerator x)) --> t
   (lessp 0 (denominator x)) --> t
```
given: (rationalp x)

```
means-rationalp
   (rationalp (rational n d)) --> t
```
given: (integerp n) and (lessp 0 d)

```
rneg-rneg
   (rneg (rneg x)) --> (reduce x)

rneg-reduce
   (rneg (reduce x)) --> (rneg x)

reduce-0
   (reduce (rational 0 x)) --> (rational 0 1)

numberp-numerator-reduce
   (numberp (numerator (reduce x))) --> (numberp (numerator (fix-rational x)))

reduce-nrationalp
   (reduce x) --> (rational 0 1)
```
given: (not (rationalp x))

```
rplus-reduce-arg2
   (rplus x (reduce y)) --> (rplus x y)

rplus-reduce-arg1
   (rplus (reduce x) y) --> (rplus x y)

requal-x-x
   (requal x x) --> t

rplus-requal-arg1
   (requal (rplus x y) (rplus z y)) --> t
```
given: (requal x z)

```
commutativity-of-rplus
   (rplus x y) --> (rplus y x))

reduce-reduce
   (reduce (reduce x)) --> (reduce x)

requal-reduce2
   (requal x (reduce y)) --> (requal x y)

requal-reduce1
   (requal (reduce x) y) --> (requal x y)

commutativity-of-requal
   (requal x y) --> (requal y x)

rational-generalization
   adds facts about rationalp's when a rational is generalized

fix-rational-rmagnitude
   (fix-rational (rmagnitude x)) --> (rmagnitude x)

fix-rational-rquotient
   (fix-rational (rquotient x)) --> (rquotient x)

fix-rational-rdifference
   (fix-rational (rdifference x)) --> (rdifference x)

fix-rational-rneg
   (fix-rational (rneg x)) --> (rneg x)
```

```
fix-rational-rtimes
   (fix-rational (rtimes x)) --> (rtimes x)

fix-rational-fix-rational
   (fix-rational (fix-rational x)) --> (fix-rational x)

fix-rational-rplus
   (fix-rational (rplus x y)) --> (rplus x y)

fix-rational-reduce
   (fix-rational (reduce x)) --> (reduce x)

rationalp-rmagnitude
    (fix-rational (rmagnitude x y)) --> (rmagnitude x y)

rationalp-rquotient
    (fix-rational (rquotient x y)) --> (rquotient x y)

rationalp-rdifference
     (fix-rational (rdifference x y)) --> (rdifference x y)

rationalp-rneg
    (fix-rational (rneg x)) --> (rneg x)

rationalp-rtimes
    (rationalp (rtimes x y)) --> t

rationalp-fix-rational
    (rationalp (fix-rational x)) --> t

rationalp-rplus
     (rationalp (rplus x y)) --> t

rationalp-reduce
     (rationalp (reduce x)) --> t

commutativity2-of-rtimes
     (rtimes x (rtimes y z)) -->  (rtimes y (rtimes x z))

rdifference
     (rdifference x y) --> (rplus x (rneg y))

correctness-of-cancel-rplus
     (rplus a (rplus b (rneg a))) --> (reduce b)
```

We define the rationals library using an NQTHM deftheory event.

```
(deftheory r2 {list of names above})
```

2.3 An example Use of R2

The automatic proof of the following problem was posed as a challenge problem. (Thanks to Matt Kaufmann who got this problem from Bill Pase at a recent conference.) The NQTHM prover proves the theorem in about 30 seconds using the rationals library. (The output of the theorem-prover when constructing this proof is Appendix A.)

```
square(x) = x * x

four_squares (a, b, c, d) = square(a) + square(b) + square(c) + square(d)


Prove:

four_squares (a, b, c, d) * four_squares (r, s, t, u)
 =
four_squares (a*r + b*s + c*t + d*u, a*s + -b*r + c*u + -d*t,
             a*t + -b*u + -c*r + d*s, a*u + b*t + -c*s + -d*r)
```

3. A Floating-Point System Axiomatization

In this section we present a set of axioms that describes the behavior of floating-point numbers. The axioms are formalized in the Boyer-Moore logic, and mechanically shown to be consistent. The axioms themselves are similar to sets proposed in previous FP work, especially [10].

3.1 The Axioms

Floating-point computation is modelled by axiomatizing some functions. The function FPP is axiomatized as a predicate that identifies floating-point values. The function ROUND is axiomatized to map any value to a floating-point value. FPMINIMUM is axiomatized to be the smallest non-zero positive floating-point value. FPMAXIMUM is axiomatized as the largest floating-point value. FPMINSPACE is a non-zero positive value that is smaller than the distance between any two floating-point numbers. ROUND-MIN and ROUND-MAX bound the inexactness introduced by the ROUND function.

To avoid the NQTHM notation that is difficult for non-Lisp programmers, we'll use a more-standard notation. Numerals are assumed to be of type rational, < is RLESSP, |x| means (RMAGNITUDE x), unary - means RNEG, and function application will be denoted f (args) rather than (f args).

The axioms were introduced using the CONSTRAIN mechanism described in [5]. Introducing them in this manner guarentees their consistency. (The model used with CONSTRAIN to demonstrate consistency is presented in subsection 3.2)

Here are the axioms:

```
0)  fpp (x)  -->  rationalp (x)
```

All floating-point values are rationals.

```
1)  fpp (0)
```

0 is a floating-point number.

```
2)  fpp (1)
```

1 is a floating-point number.

```
3)    rationalp (x)  -->  fpp (reduce (x)) = fpp (x)
```

If x is rational, then the reduction of x (removing common factors of the numerator and denominator) does not affect whether x is a floating-point number.

4) fpp (fpmaximum)

FPMAXIMUM is a floating-point number

5) fpp (x) --> fpmaximum >= |x|

A floating-point number is not larger in magnitude than FPMAXIMUM.

6) (fpp (x) and x <> 0) --> |x| >= fpminimum

A non-zero floating-point number's magnitude is not less than FPMINIMUM.

7) fpp (round (x))

ROUND returns a floating-point value.

8) rationalp (x) --> fpp (- x)) = fpp (x)

X is a floating-point value iff -X is.

9) fpp (y) and x >= y --> round (x) >= y

Applying ROUND to a value X no less than a floating-point value Y will not return a value less than Y.

10) fpp (y) and x <= y --> round (x) <= y

Applying ROUND to a value not greater than floating-point value Y will not return a value greater than Y.

11) x >= FPMINIMUM and x <= FPMAXIMUM --> round(x) >= (ROUND-MIN * x)

Values in range to be floating-point numbers when rounded will not be smaller than ROUND-MIN times their original value.

12) x >= FPMINIMUM and x <= FPMAXIMUM --> round(x) <= (ROUND-MAX * x)

Values in range to be floating-point numbers when rounded will not be larger than ROUND-MAX times their original value.

13) 0 <= FPMINSPACE

FPMINSPACE is a positive, non-zero value.

14) round (- x) = - (round (x))

Applying ROUND to -X yields the same value as negating the result of applying ROUND to X.

```
15)      fpp (x) and delta > 0 and delta < FPMINSPACE
         -->
         not fpp(x + delta)
```

Values within FPMINSPACE of a floating-point value not equal to that floating-point value are not floating-point values.

3.2 A Model that Shows the Axioms to be Consistent

The first thing we'd like to show with our set of axioms is that the axioms themselves are consistent. If there is a set of functions that behave in the manner proscribed by the axioms above then the axioms will have been shown to be consistent.[1]

Though the axioms are suggestive of a conventional floating-point system, assigning simple functions in the manner described below makes all the axioms true.

```
fpp(x)         (x = 0) or (x = 1) or (x = -1)

fpminspace     1

fpminimum      1

fpmaximum      1

round-max      1

round-min      1

round(x)       if |x| < 1
                  0
                if x >= 0
                   1
                 -1
```

3.3 What is a Floating-Point Program?

The NQTHM logic will be used to express programs. [4] This report contains programs using infix notation to represent terms in the logic in order to assist readers unfamiliar with the logic's Lisp-like notation. Even so, the ultimate authority about what has been proved is in the proof script that has been checked by the theorem prover.

Several axioms were presented in the previous subsection that describe floating-point operations.

[1]Though several sets of axioms for floating-point operations have been proposed in the literature, to my knowledge the consistency of a set of axioms has never been addressed. It's not hard, and it is suprising that it has not been handled explicitly before. This is typical of the kind of thing that might be skipped in a written proof but which is required in a mechanically-checked proof.

Programs that manipulate floating point values will use some of the functions described, such as **round** and **fpp**. They will also use non-floating-point functions, such as **if**.

One potential problem with using the NQTHM logic as our programming notation is that some programs that can be expressed in the logic are outside our intended domain. This is not normally a consideration, since programming languages with floating point operations typically exclude unreasonable statements. For example, a program that performs an exact multiplication of two floating-point values is not possible to express in most programming languages, and appears to violate the spirit of what we mean by a "floating-point" program. Precisely what NQTHM functions would commonly be considered floating-point programs is open to debate since what is "generally considered a floating-point program" is not a formal specification for deciding what is a floating-point program and what is not.

For our purposes, the following conditions will be sufficient to claim that a program is a floating-point program.

- Every instance of **rplus**, **rdifference**, and **rtimes** is applied to arguments that are provably **fpp**
- Every instance of **rquotient** is applied to arguments that are provably **fpp** or has a provably **fpp** first argument and a rational constant power of two second argument.
- Every instance of **rplus**, **rdifference**, **rtimes**, and **rquotient** is immediately surrounded by a **round** function.
- Every instance of **rmagnitude** and **rneg** is applied to an argument that is provably **fpp**.
- Every instance of **rlessp** and **requal** is applied to arguments that are provably **fpp**.
- Every instance of a rational operation other than the eight already mentioned - **rplus**, **rdifference**, **rquotient**, **rtimes**, **rneg**, **rmagnitude**, **rlessp**, and **requal** - will be translatable into an operation composed only of those eight that meets the previously-described restrictions and functions commonly built-in to programming languages.
- Every function used in the program is either a floating-point function or a function commonly built-in to programming languages (such as **if**).

Note that the property of being a floating-point program is not purely syntactic since an argument to a rational function may need to be proved to be a floating-point value.

4. An Example Floating-Point Program

In this section we develop a program that finds a zero of an arbitrary floating-point function. We wish a zero-finding program to return a pair of floating-point values that bound a small region containing a zero. This means that the function applied to the two endpoints returns values with opposite signs.

We will solve the zero-searching problem by writing a floating-point program that checks to see if the current region is small enough, and if it isn't finds a midpoint of the region and recursively searches one of the two subregions.

The function **find-func-zero** will look like this:

```
  find-func-zero (a,b)
=
  if unreasonable (a,b)
    0
   if close (a,b)
     (a,b)
     let mid := fp-mid (a,b)
        if sign (func (mid)) = sign (func (a))
          find-func-zero (mid,b)
         find-func-zero (a,mid)
```

The following subsections define the functions **func**, **fp-mid**, **close**, and **unreasonable**. **fp-mid** is really a family of search programs that works for different **func**s, and different values of the floating-point constants. Properties we require we add as axioms, using CONSTRAIN events to ensure consistency.

4.1 func

Since **find-func-zero** is supposed to find a zero of an arbitrary floating-point function **func**, we should specify the function **func** as weakly as possible. We can do this by adding the axiom that **func** returns a floating-point value, and not defining precisely what it returns. To insure consistency, we witness the function **func** by the function that returns a floating-point value of 0.

Axiom: `func(x) is a floating-point value.`

4.2 fp-mid

There are several programming choices we can make for finding the midpoint of a region. We will use a very obvious program, namely adding the two values and dividing by 2. Other choices would eliminate the problem of floating-point overflow. For example, if the two values were each divided by 2 and then added together, overflow could not occur. Our choice will limit the applicability of the program slightly, but our example will demonstrate that the most-obvious programming solution can be specified and proved correct based upon the floating-point axioms.

The axiomatization of floating-point operations suggests another potential problem with this program. The values might be small enough that floating-point-adding them together and exactly-dividing by 2 will yield a value less in absolute value than FPMINIMUM. If this happens, the result is not guaranteed to fall between the two endpoints. By the axioms of floating-point, we can prove that this will not occur if the smaller (in absolute value) endpoint is at least FPMINIMUM/ROUND-MIN (in absolute value.) We wish to work with floating-point values, so since this value is important we axiomatize a constant to be a floating-point value at least as large as this value. The constant will be called MID-BOUND2.

proposed axiom:
```
fpp(MID-BOUND2)
  and
MID-BOUND2 > FPMINIMUM/ROUND-MIN
```

We wish to witness this constant to show that adding an axiom like the one above does not cause an inconsistency. Unfortunately, our lemmas so far do not guarantee that MID-BOUND2 as described above actually exists. It is consistent with our axioms of floating-point that FPMAXIMUM = FPMINIMUM, in which case MID-BOUND2 does not exist. It therefore is necessary to weaken this axiom, and we add the following instead.

Axiom:
```
fpp(MID-BOUND2)
and
  (MID-BOUND2 >= FPMINIMUM/ROUND-MIN)
  or
  (FPMAXIMUM < FPMINIMUM/ROUND-MIN)
```

MID-BOUND2 can now be witnessed with the constant FPMAXIMUM, which insures that we have not added a contradiction by adding the axiom above.

To aid in our eventual proof, we define **fp-mid** to have several cases. Also, we'll write this program with the implicit assumption that a <= b.

```
fp-mid (a b)
=
if (a < MID-BOUND2) and (b > MID-BOUND2)
  MID-BOUND2
 if (a < 0) and (b > 0)
   0
  if (a < - MID-BOUND2) and (b > - MID-BOUND2)
   - MID-BOUND2
   ROUND (ROUND (a + b) / 2)
```

4.3 close

close(a,b) should return true if it is not guaranteed that a < **fp-mid**(a,b) < b. Otherwise it should return false so that **find-func-zero** returns the smallest range possible.

From the previous discussion, one case where **close**(a,b) should return true is if 0 <= a <= b <= MID-BOUND2, or - MID-BOUND2 <= a <= b <= 0. (This is not the strictest possible bound, but it is close and the infrequent gain in performance from using the most-strict bound has been judged to be not worth the much greater complexity.)

We can derive a closeness bound for values not close to 0 from the FP axioms too. We will call this bound MID-BOUND and axiomatize it in a manner similar to MID-BOUND2. The complex terms in this axiom correspond to the inexactness introduced by applying **fp-mid**.[2]

```
Axiom:   fpp (MID-BOUND)
       and
         [fpp (x)
          and
          non-negative (x)
          and
          FPMAXIMUM >= (x * 2)]
         -->
         [((2 - (ROUND-MAX * ROUND-MAX)) * x) / (ROUND-MAX * ROUND-MAX)
               >= ROUND (MID-BOUND * x)
          and
          ((ROUND-MIN * ROUND-MIN) * x) / (2 - (ROUND-MIN * ROUND-MIN))
               >= ROUND (MID-BOUND * x)]
```

MID-BOUND may be witnessed by 0 to show that the axiom does not cause an inconsistency.

We may now define **close**(a,b).

[2]Facts about the bounds introduced in this axiom are proved. They are derived by applying axioms 11 and 12 from section 3 to the function **fp-mid**.

```
close(a,b)
=
  a >= 0
  and
    a >= round (b * MID-BOUND)
    or
    b <= MID-BOUND2
or
  b <= 0
  and
    b >= round (a * MID-BOUND)
    or
    a >= - MID-BOUND2
```

4.4 unreasonable

unreasonable(a,b) is true if and only if an invariant assumed by another part of the program is violated. The simplest such invariants that we've discussed are that a and b must be **fpp** and a <= b.

As described previously, it is consistent with the FP axioms that FPMAXIMUM < FPMINIMUM/ROUND-MIN. A realistic floating-point system would not have this property, so we will add its negation to the definition of **unreasonable** and be willing later to accept a correctness theorem that has this unimportant restriction.

Our earlier choice of the midpoint program made the program vulnerable to "floating-point overflow". This means that a floating-point operation takes place whose corresponding exact result is larger in absolute value than FPMAXIMUM. In this example overflow of the midpoint addition operation might cause the midpoint program to return a value not strictly between the endpoints. Our program is therefore not guaranteed to work "correctly" if |a + b| > FPMAXIMUM. (+ here is exact addition.) A somewhat overly-restrictive but sufficient condition that would assure no overflow is |a| <= FPMAXIMUM/2 and |b| <= FPMAXIMUM/2. Since we wish to restrict ourselves to FP numbers in our program to test for this condition, we'll axiomatize an fp value to be less than FPMAXIMUM/2.

```
Axiom:   fpp (MID-BOUND3)
        and
          MID-BOUND3 <= FPMAXIMUM/2
```

We're now in a position to define **unreasonable**.

```
unreasonable(a,b)
=
not fpp(a)  or  not fpp(b)  or  a > b
  or
FPMAXIMUM < FPMINIMUM/ROUND-MIN
  or
RMAGNITUDE(a) > MID-BOUND3  or  RMAGNITUDE(b) > MID-BOUND3
```

4.5 Is find-func-zero a Floating-Point Program?

In the section on axiomatizing floating-point, sufficient conditions for calling a program a floating-point program were stated. Inspection of **find-func-zero** shows that it meets these conditions in all but one respect. FPMINIMUM/ROUND-MIN is used in an **rlessp** term, and there is no guarantee that this value is **fpp**. However, the entire offending term (described above in the description of **unreasonable**) is a boolean constant for any particular floating-point system. That is, a given floating-point system will have numerical values assigned to FPMINIMUM, ROUND-MIN, and FPMAXIMUM, so the truth of the term can be determined before runtime. This violation of one of our sufficient conditions therefore does not cause the program to fail to be a floating-point program.

Thus, **find-func-zero** can sensibly be called an example of a floating-point program.

5. A Floating-Point Program Correctness Theorem

This section makes an informal argument that the zero-finding program works and presents the PC-NQTHM events that represent the zero-finding program and a mechanically-checked statement about its correctness.

5.1 Correctness Argument Outline

The informal correctness theorem is:

```
not unreasonable (a,b)
and
sign (func (a)) <> sign (func (b))
-->
let (lower, upper) := find-func-zero (a,b)
  (sign (func (lower)) <> sign (func (upper))
   and
   close (lower, upper))
```

The correctness proof of the zero-finding program has many details. The proof script checked by the prover (not including supporting libraries about arithmetic, rationals, and floating-point numbers) is 169,000 bytes long. The following list of lemmas outlines the essential argument. (The name of the corresponding PC-NQTHM events the proof script are given in parenthesis.)

1. If not **unreasonable**(a,b) and not **close**(a,b), then a < fp-mid(a,b) < b (**fp-mid-fact1**, **fp-mid-fact2**)
2. **fp-mid**(a,b) returns a floating-point value (**fpp-mid**)
3. **find-func-zero-measure**(a,b,l) expects 3 rational numbers and returns the greatest natural number n such that (b - a) >= (n * l) (**find-func-zero-measure**)
4. If x, y, and m are **fpp**, and x < m < y, then **find-func-zero-measure**(a,b,FPMINSPACE) is reduced if a or b is replaced by m (**fpp-means-find-func-ok**)
5. **find-func-zero** terminates (from previous lemmas)
6. **reasonable**(a,b) --> **reasonable**(a , fp-mid(a,b)) and **reasonable**(**fp-mid**(a,b) , b)
7. **sign**(**func**(a)) <> **sign**(**func**(b)) --> **sign**(**func**(lower)) <> **sign**(**func**(upper)) *[where (lower,upper* := **find-func-zero**(a,b)])
8. By induction on **find-func-zero** (justified by lemma 5) and lemmas 6 and 7, the correctness theorem holds. (**find-func-zero-returns-a-zero**,**find-func-zero-returns-close-values**)

5.2 An NQTHM Zero-Finding Program and Correctness Theorem

The zero-finding program and its correctness proof have been formalized in the NQTHM logic. [4] The proof checker enhancement of the theorem prover [12] has accepted the program and its proof. Axiomatized functions were added using constrain events that have been proved to keep the prover state consistent. [5]

The following events have been accepted by the theorem prover. Their acceptance constitutes a mechanical proof of the correctness of the program. (Theorem prover hints have been deleted from the events.)

The zero-finding program is expressed in the logic as

```
(defn fp-mid (x y)
  (if (and (rlessp x (mid-bound2))
           (rlessp (mid-bound2) y))
      (mid-bound2)
    (if (and (rlessp x (rational 0 1))
             (rlessp (rational 0 1) y))
        (rational 0 1)
      (if (and (rlessp x (rneg (mid-bound2)))
               (rlessp (rneg (mid-bound2)) y))
          (rneg (mid-bound2))
        (round (rquotient (round (rplus x y))
                          (rational 2 1)))))))

(defn find-func-zero (a b)
  (if (or (not (rlessp a b))                          ; not reasonable(a,b)
          (rlessp (mid-bound3) (rmagnitude a))
          (rlessp (mid-bound3) (rmagnitude b))
          (rlessp (fpmaximum)
                  (rquotient (fpminimum) (round-min)))
          (not (fpp a))
          (not (fpp b)))
      (rational 0 1)
    (if (or (and (numberp (numerator a))              ; close (a,b)
                 (or (not (rlessp a
                                  (round (rtimes b (mid-bound1)))))
                     (not (rlessp (mid-bound2) b))))
            (and (or (rzerop b)
                     (negativep (numerator b)))
                 (or (not (rlessp (round (rtimes a (mid-bound1)))
                                  b))
                     (not (rlessp (mid-bound2) (rneg a))))))
        (cons a b)
      (let ((mid (fp-mid a b)))
        (if (equal (numberp (numerator (func a)))
                   (numberp (numerator (func mid))))
            (find-func-zero mid b)
          (find-func-zero a mid))))))
```

The correctness theorem is expressed as two NQTHM events.

```
(prove-lemma find-func-zero-returns-close-values (rewrite)
             (implies
              (and                                  ; reasonable(a,b)
               (rlessp a b)
               (not (rlessp (mid-bound3) (rmagnitude a)))
               (not (rlessp (mid-bound3) (rmagnitude b)))
               (not (rlessp (fpmaximum)
                            (rquotient (fpminimum) (round-min))))
               (fpp a)
               (fpp b))
              (let ((lower (car (find-func-zero a b)))
                    (upper (cdr (find-func-zero a b))))
                (or (and (numberp (numerator lower))      ; close (lower,upper)
                         (or (not (rlessp lower
                                          (round (rtimes upper (mid-bound1)))))
                             (not (rlessp (mid-bound2) upper))))
                    (and (or (rzerop upper)
                             (negativep (numerator upper)))
                         (or (not (rlessp (round (rtimes lower (mid-bound1)))
                                          upper))
                             (not (rlessp (mid-bound2) (rneg lower)))))))))
```

```
(prove-lemma find-func-zero-returns-a-zero (rewrite)
       (implies
         (and (not (equal (numberp (numerator (func a)))
                          (numberp (numerator (func b)))))
              (rlessp a b)                                  ; reasonable(a,b)
              (not (rlessp (mid-bound3) (rmagnitude a)))
              (not (rlessp (mid-bound3) (rmagnitude b)))
              (not (rlessp (fpmaximum)
                           (rquotient (fpminimum) (round-min))))
              (fpp a)
              (fpp b))
        (not (equal (numberp (numerator (func (car (find-func-zero a b)))))
                    (numberp (numerator (func (cdr (find-func-zero a b)))))))))
```

6. Future Work

This report describes work in an ongoing project whose ultimate goal is to create in a realistic floating-point number system a substantial floating-point program with a mechanically-checked correctness proof. The mid-point example presented here appears to be the first mechanically-checked floating-point program, and is a step toward that goal.

6.1 Planned Work

Much work is planned.

Proofs about floating-point programs require facts about rational arithmetic operations. The rational library described in section 2 will be greatly enhanced and will allow much faster development of proofs. The development of a useful library of rational number facts is an interesting challenge that may have application beyond proofs about floating point arithmetic.

The floating-point system described in section 3 is less-complex than the floating-point system found on most computers. A realistic floating-point system will be formalized to allow proofs about floating point operations.

The system in section 3 was shown to have a trivial model so as to demonstrate the consistency of the axioms of its formalization. The enhanced floating-point system that will be constructed will be proved to have a model that encompasses a significant portion of the IEEE standard for floating-point arithmetic. [11] This will insure that the axioms describe a consistent system that is realistic. Also, any applications proved correct will be able to execute using the model.

An application program that uses the enhanced floating-point formalization will be proved correct. The example that will probably be pursued is a correctness proof of a program that calculates the sine function. [7] The correctness theorem tightly bounds the forward error of the calculation.

6.2 Other Possible Work

The planned work suggests several possible interesting detours.

The floating-point system model could actually be implemented. This entails writing a compiler for a portion of the logic that includes floating point operations. The target language of the compiler would probably be Piton [16] so that the resulting code could be run on the verified stack. [3] The appeal of verifying a floating-point program down to the level of hardware is very strong.

It appears that backward error analysis of floating-point programs [18] might benefit from algorithms that automate finding the consistency of sets of inequalities. Such a theorem prover might quickly verify the correctness of the statement of backward error theorems. This possibility will be further explored, and if it seems fruitful such a system will be built.

The techniques developed to verify the floating-point program will hopefully be applicable to other examples. Other programs may be verified to evaluate the general effectiveness of the approach.

Appendix A
An example R2 proof

This appendix presents the output of the theorem prover when given the example problem in subsection 2.3.

```
(DEFN SQUARE (X) (RTIMES X X))

     Note that (RATIONAL-FORMP (SQUARE X)) is a theorem.


[ 0.0 0.0 0.1 ]

SQUARE

(DEFN FOUR-SQUARES
      (A B C D)
      (RPLUS (SQUARE A)
             (RPLUS (SQUARE B)
                    (RPLUS (SQUARE C) (SQUARE D)))))

     From the definition we can conclude that:
      (RATIONAL-FORMP (FOUR-SQUARES A B C D))
is a theorem.


[ 0.1 0.0 0.0 ]

FOUR-SQUARES

(LEMMA FOUR-SQ NIL
       (EQUAL (RTIMES (FOUR-SQUARES A B C D)
                      (FOUR-SQUARES R S T0 U))
              (FOUR-SQUARES (RPLUS (RTIMES A R)
                                   (RPLUS (RTIMES B S)
                                          (RPLUS (RTIMES C T0) (RTIMES D U))))
                            (RPLUS (RTIMES A S)
                                   (RPLUS (RNEG (RTIMES B R))
                                          (RPLUS (RTIMES C U)
                                                 (RNEG (RTIMES D T0)))))
                            (RPLUS (RTIMES A T0)
                                   (RPLUS (RNEG (RTIMES B U))
                                          (RPLUS (RNEG (RTIMES C R))
                                                 (RTIMES D S))))
                            (RPLUS (RTIMES A U)
                                   (RPLUS (RTIMES B T0)
                                          (RPLUS (RNEG (RTIMES C S))
                                                 (RNEG (RTIMES D R)))))))
       ((ENABLE-THEORY R2)
        (ENABLE FOUR-SQUARES SQUARE)))
```

```
This formula can be simplified, using the abbreviation FOUR-SQUARES, to:

      (EQUAL
       (RTIMES (RPLUS (SQUARE A)
                      (RPLUS (SQUARE B)
                             (RPLUS (SQUARE C) (SQUARE D))))
               (RPLUS (SQUARE R)
                      (RPLUS (SQUARE S)
                             (RPLUS (SQUARE T0) (SQUARE U)))))
       (RPLUS
        (SQUARE (RPLUS (RTIMES A R)
                       (RPLUS (RTIMES B S)
                              (RPLUS (RTIMES C T0) (RTIMES D U)))))
        (RPLUS (SQUARE (RPLUS (RTIMES A S)
                              (RPLUS (RNEG (RTIMES B R))
                                     (RPLUS (RTIMES C U)
                                            (RNEG (RTIMES D T0))))))
               (RPLUS (SQUARE (RPLUS (RTIMES A T0)
                                     (RPLUS (RNEG (RTIMES B U))
                                            (RPLUS (RNEG (RTIMES C R))
                                                   (RTIMES D S)))))
                      (SQUARE (RPLUS (RTIMES A U)
                                     (RPLUS (RTIMES B T0)
                                            (RPLUS (RNEG (RTIMES C S))
                                                   (RNEG (RTIMES D R)))))))))),

which simplifies, applying the lemmas ASSOCIATIVITY-OF-RTIMES,
RTIMES-RPLUS-ARG1, COMMUTATIVITY2-OF-RTIMES, RTIMES-RPLUS-ARG2,
ASSOCIATIVITY-OF-RPLUS, COMMUTATIVITY2-OF-RPLUS, COMMUTATIVITY-OF-RTIMES,
COMMUTATIVITY-OF-RPLUS, RNEG-RPLUS, RNEG-RNEG, REDUCE-RTIMES, RTIMES-RNEG-ARG2,
RTIMES-RNEG-ARG1, and CORRECTNESS-OF-CANCEL-RPLUS, and unfolding the
definition of SQUARE, to:

      T.

Q.E.D.


[ 1.8 34.7 0.2 ]

FOUR-SQ
T

>
```


Appendix B
The Rational Library

This appendix lists the forms that create the rational number library. Some of the events use proof-checker instructions as hints to the prover [12]. These hints have been removed from this listing in the interest of space.

```
(note-lib "/disk1/home/bevier/libs/integers")
(load "/usr/home/bevier/nqthm-init.lsp")
(load "/usr/home/wilding/numerical/arithmeticmods.events")
; ---------------------------------------------------------------------------
; Rational Numbers
; ---------------------------------------------------------------------------

(add-shell rational nil rational-formp
           ((numerator   (one-of numberp negativep) zero)
            (denominator (one-of numberp) zero)))

(deftheory rational-defns
 (COUNT-RATIONAL NUMERATOR-DENOMINATOR-ELIM
    RATIONAL-NUMERATOR-DENOMINATOR RATIONAL-EQUAL DENOMINATOR-LESSEQP
    DENOMINATOR-LESSP DENOMINATOR-TYPE-RESTRICTION
    DENOMINATOR-NRATIONAL-FORMP DENOMINATOR-RATIONAL NUMERATOR-LESSEQP
    NUMERATOR-LESSP NUMERATOR-TYPE-RESTRICTION
    NUMERATOR-NRATIONAL-FORMP NUMERATOR-RATIONAL
    DENOMINATOR NUMERATOR *1*RATIONAL-FORMP
    RATIONAL-FORMP))

(definition rationalp (x)
  (and (rational-formp x)
       (integerp (numerator x))
       (not (zerop (denominator x)))))

(definition fix-rational (x)
  (if (rationalp x) x (rational 0 1)))

(definition reduce (r)
  (if (rationalp r)
      (if (negativep (numerator r))
          (rational (minus (quotient (negative-guts (numerator r))
                                     (gcd (negative-guts (numerator r))
                                          (denominator r))))
                    (quotient (denominator r)
                              (gcd (negative-guts (numerator r))
                                   (denominator r))))
        (rational (quotient (numerator r) (gcd (numerator r) (denominator r)))
                  (quotient (denominator r) (gcd (numerator r)
                                                 (denominator r)))))
    (rational 0 1)))

(definition simple-rplus (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (rational (iplus (itimes (numerator a) (denominator b))
                     (itimes (numerator b) (denominator a)))
              (itimes (denominator a) (denominator b)))))
```

```
(definition rplus (x y)
  (reduce (simple-rplus x y)))

(definition simple-rneg (x)
  (let ((a (fix-rational x)))
    (rational (ineg (numerator a))
              (denominator a))))
(definition rneg (x)
  (reduce (simple-rneg x)))

(definition rdifference (a b)
  (rplus a (rneg b)))

(definition simple-rtimes (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (rational (itimes (numerator a) (numerator b))
              (times (denominator a) (denominator b)))))

(definition rtimes (x y)
  (reduce (simple-rtimes x y)))
#|
(definition simple-rquotient (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (if (negativep (numerator a))
        (if (negativep (numerator b))
            (rational (times (negative-guts (numerator a)) (denominator b))
                      (times (negative-guts (numerator b)) (denominator a)))
        (rational (minus (times (negative-guts (numerator a)) (denominator b)))
                  (times (numerator b) (denominator a))))
      (if (negativep (numerator b))
          (rational (minus (times (numerator a) (denominator b)))
                    (times (negative-guts (numerator b)) (denominator a)))
        (rational (times (numerator a) (denominator b))
                  (times (numerator b) (denominator a)))))))

(definition rquotient (x y)
  (reduce (simple-rquotient x y)))
|#

(definition rzerop (x)
  (or
   (not (rationalp x))
   (equal (numerator x) 0)))


(defn simple-rinverse (r)
  (if (rzerop r)
      (rational 0 1)
  (if (negativep (numerator r))
     (rational (ineg (denominator r))
               (ineg (numerator r)))
    (rational (denominator r) (numerator r)))))
```

```
(defn rinverse (r)
  (reduce (simple-rinverse r)))

(definition rquotient (x y)
  (rtimes x (rinverse y)))

(definition simple-rmagnitude (x)
  (let ((a (fix-rational x)))
    (if (negativep (numerator a))
        (rneg a)
      a)))

;(definition simple-rmagnitude (x)
;  (let ((a (fix-rational x)))
;    (if (negativep (numerator a))
;       (rational (negative-guts (numerator a)) (denominator a))
;      a)))

(definition rmagnitude (x)
  (reduce (simple-rmagnitude x)))

(defn rlessp (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (ilessp (itimes (numerator a) (denominator b))
            (itimes (numerator b) (denominator a)))))

(defn requal (x y)
  (let ((a (fix-rational x)) (b (fix-rational y)))
    (equal (itimes (numerator a) (denominator b))
           (itimes (numerator b) (denominator a)))))
;;;;;;


(prove-lemma integerp-minus (rewrite)
             (equal (integerp (minus x))
                    (lessp 0 x))
             ((enable integerp)))


(prove-lemma fix-int-on-integers (rewrite)
             (implies
              (integerp x)
              (equal (fix-int x) x))
             ((enable fix-int)))

(prove-lemma itimes-ineg-arg1 (rewrite)
             (equal
              (itimes (ineg x) y)
              (ineg (itimes x y)))
             ((enable itimes ineg)))

(prove-lemma itimes-ineg-arg2 (rewrite)
             (equal
              (itimes x (ineg y))
              (ineg (itimes x y)))
             ((enable itimes ineg)))

(prove-lemma integerp-if-negativep-non-zero (rewrite)
             (implies
              (and
               (negativep x)
               (not (equal x (minus 0))))
              (integerp x))
             ((enable integerp)))
```

```
(prove-lemma integerp-if-numberp (rewrite)
             (implies
              (numberp x)
              (integerp x))
             ((enable integerp)))


(prove-lemma itimes-negativep-arg1 (rewrite)
             (implies
              (negativep x)
              (equal
               (itimes x y)
               (ineg (itimes (negative-guts x) y))))
             ((enable ineg itimes)))

(prove-lemma itimes-negativep-arg2 (rewrite)
             (implies
              (negativep y)
              (equal
               (itimes x y)
               (ineg (itimes x (negative-guts y)))))
             ((enable itimes ineg)))


(prove-lemma equal-ineg-ineg (rewrite)
             (implies
              (and
               (integerp x)
               (integerp y))
              (equal
               (equal (ineg x) (ineg y))
               (equal x y)))
             ((enable integerp ineg)))

(prove-lemma itimes-is-times (rewrite)
             (implies
              (and
               (numberp x)
               (numberp y))
              (equal (itimes x y) (times x y)))
             ((enable itimes)))

(prove-lemma iplus-is-plus (rewrite)
             (implies
              (and
               (numberp x)
               (numberp y))
              (equal (iplus x y) (plus x y)))
             ((enable iplus)))


;;;;;;;;


(prove-lemma rationalp-reduce (rewrite)
             (rationalp (reduce x)))

(prove-lemma rationalp-rplus (rewrite)
             (rationalp (rplus x y))
             ((disable reduce rationalp)))

(prove-lemma rationalp-fix-rational (rewrite)
             (rationalp (fix-rational x)))

(prove-lemma rationalp-rtimes (rewrite)
             (rationalp (rtimes x y))
             ((disable reduce rationalp)))

(prove-lemma rationalp-rneg (rewrite)
             (rationalp (rneg x))
             ((disable reduce rationalp)))

(prove-lemma rationalp-rdifference (rewrite)
             (rationalp (rdifference x y))
             ((disable rplus rationalp rneg)))

(prove-lemma rationalp-rquotient (rewrite)
             (rationalp (rquotient x y))
             ((disable reduce rationalp)))

(prove-lemma rationalp-rmagnitude (rewrite)
             (rationalp (rmagnitude x))
             ((disable rationalp reduce)))


(prove-lemma fix-rational-reduce (rewrite)
             (equal (fix-rational (reduce x))
                    (reduce x))
             ((disable reduce)))

(prove-lemma fix-rational-rplus (rewrite)
             (equal (fix-rational (rplus x y))
                    (rplus x y))
             ((disable reduce)))

(prove-lemma fix-rational-fix-rational (rewrite)
             (equal (fix-rational (fix-rational x))
                    (fix-rational x)))

(prove-lemma fix-rational-rtimes (rewrite)
             (equal (fix-rational (rtimes x y))
                    (rtimes x y))
             ((disable reduce)))

(prove-lemma fix-rational-rneg (rewrite)
             (equal (fix-rational (rneg x))
                    (rneg x))
             ((disable reduce)))

(prove-lemma fix-rational-rdifference (rewrite)
             (equal (fix-rational (rdifference x y))
                    (rdifference x y))
             ((disable reduce)))

(prove-lemma fix-rational-rquotient (rewrite)
             (equal (fix-rational (rquotient x y))
                    (rquotient x y))
             ((disable reduce)))

(prove-lemma fix-rational-rmagnitude (rewrite)
             (equal (fix-rational (rmagnitude x))
                    (rmagnitude x))
             ((disable reduce)))
```

```
(prove-lemma rational-generalization (generalize)
             (and
              (implies
               (rationalp x)
               (integerp (numerator x)))
              (implies
               (rationalp x)
               (numberp (denominator x)))
              (implies
               (rationalp x)
               (not (zerop (denominator x))))))



;;;; some divides and gcd facts

;(prove-lemma remainder-times-fact1 (rewrite)
;            (implies
;             (equal (times a b) (times c d))
;              (equal (remainder (times c d) a) 0)))


(prove-lemma gcd-remainder-fact1 (rewrite)
             (implies
              (and
               (lessp 1 b)
               (equal (gcd a b) 1))
              (not (equal (remainder a b) 0)))
             ((enable gcd)))

(lemma gcd-remainder-fact2 (rewrite)
       (implies
        (and
         (lessp 1 b)
         (equal (gcd b a) 1))
        (not (equal (remainder a b) 0)))
       ((enable commutativity-of-gcd gcd-remainder-fact1)))

(defn gcd-times1-induct (x y)
  (if (zerop y)
      t
    (gcd-times1-induct x (sub1 y))))

(prove-lemma gcd-times1 (rewrite)
             (equal
              (gcd x (times x y))
              (fix x))
             ((induct (gcd-times1-induct x y))))

(lemma gcd-times2 (rewrite)
       (equal
        (gcd y (times x y))
        (fix y))
       ((disable fix)
        (enable commutativity-of-times)
        (use (gcd-times1 (x y) (y x)))))

(prove-lemma gcd-quotient-quotient
             (rewrite)
             (implies (and (lessp 0 a) (lessp 0 b))
                      (equal (gcd (quotient a (gcd a b))
                                  (quotient b (gcd a b)))
                             1)))
```

```
;(lemma gcd-quotient-quotient-2 (rewrite)
;       (implies
;       (lessp 0 a)
;       (equal (gcd (quotient a (gcd a b))
;                   (quotient b (gcd a b)))
;             1))
;       ((use (gcd-quotient-quotient))
;       (enable-theory arithmetic)))

(prove-lemma divides-each-equality (rewrite)
             (equal
              (and
               (equal (remainder a b) 0)
               (equal (remainder b a) 0))
              (equal (fix a) (fix b))))

;;;;;;
;;;;; Following until gcd-remainder-times-fact1-proof done with matt k.
;;;;;;
(disable IDIFFERENCE-IPLUS-CANONICALIZER1)
(enable ilessp)
(enable idifference)
(enable iplus)
(enable itimes)
(enable ineg)
(enable integerp)
(enable fix-int)
(enable izerop)



(defn gcd-factors (x y)
  ;; returns a and b s.t. a*x+b*y=gcd.  Assumes that x and y are non-zero.
  (cond ((zerop x)
         ;; 0*0+1*y=y, which is the gcd of 0 and y
         (cons 0 1))
        ((zerop y)
         ;; 1*x+0*y=x, which is the gcd of x and 0
         (cons 1 0))
        ((lessp x y)
         ;; if a*x+b*(y-x) = gcd then (a-b)*x+b*y = gcd
         (let ((factors (gcd-factors x (difference y x))))
           (cons (idifference (car factors) (cdr factors))
                 (cdr factors))))
        (t
         ;; so (leq y x)
         ;; if a*(x-y)+b*y = gcd then a*x+(b-a)*y = gcd
         (let ((factors (gcd-factors (difference x y) y)))
           (cons (car factors)
                 (idifference (cdr factors) (car factors))))))
  ((ord-lessp (cons (add1 x) (fix y)))))

(prove-lemma gcd-factors-gives-linear-combination ()
  (let ((factors (gcd-factors x y)))
    (let ((a (car factors))
          (b (cdr factors)))
      (implies (and (numberp x) (numberp y))
               (equal (iplus (itimes a x) (itimes b y))
                      (gcd x y))))))


(enable iremainder)
(enable iquotient)
```

```
(lemma gcd-factors-gives-linear-combination-rewrite (rewrite)
  (let ((factors (gcd-factors x y)))
    (let ((a (car factors))
          (b (cdr factors)))
      (implies (and (numberp x) (numberp y))
               (equal (gcd x y)
                      (iplus (itimes a x) (itimes b y))))))
  ((use (gcd-factors-gives-linear-combination))))

#|
ax+by = 1
c(ax+by) = c
cax+cby = c
cax + b(cy) = c
cax + b(qx) = c where q = (quotient cy x)
x(ca + bq) = c
so x divides c
|#

(lemma dpr-hack1 (rewrite)
  (equal
   (itimes x (iplus (itimes c a) (itimes b q)))
   (iplus (itimes c (itimes a x))
                 (itimes b (itimes q x))))
  ((enable-theory integers)))

(prove-lemma dpr-hack2 (rewrite)
  (let ((q (iquotient (itimes c y) x)))
    (implies (and (numberp x)
                  (numberp y)
                  (numberp c)
                  (equal (iremainder (itimes c y) x) 0))
             (equal (itimes q x)
                    (itimes c y)))))

(lemma dpr-hack3 (rewrite)
  (equal (iplus (itimes c (itimes a x))
                (itimes b (itimes c y)))
         (iplus (itimes c (itimes a x))
                (itimes c (itimes b y))))
  ((enable-theory integers)))

(lemma dpr-hack4 (rewrite)
  (equal (iplus (itimes c (itimes a x))
                (itimes c (itimes b y)))
         (itimes c (iplus (itimes a x)
                          (itimes b y))))
  ((enable-theory integers)))

(lemma dpr-hack5 (rewrite)
  (implies (and (numberp x)
                (numberp y)
                (numberp c)
                (equal (iremainder (itimes c y) x) 0))
           (equal (itimes x (iplus (itimes c a)
                                   (itimes b (iquotient (itimes c y) x))))
                  (itimes c (iplus (itimes a x)
                                   (itimes b y)))))
  ((enable dpr-hack1 dpr-hack2 dpr-hack3 dpr-hack4)))

(prove-lemma remainder-0-sufficiency (rewrite)
  (implies (and
            (numberp x)
            (numberp c)
            (equal (itimes x p) c))
           (equal (equal (remainder c x) 0) t)))
```

```
(prove-lemma divides-product-reduction
    (rewrite)
    (implies (and (numberp x)
                  (numberp y)
                  (numberp c)
                  (equal (gcd x y) 1)
                  (equal (remainder (times c y) x) 0))
             (equal (remainder c x) 0))
    )

(disable remainder-0-sufficiency)
(disable dpr-hack1)
(disable dpr-hack2)
(disable dpr-hack3)
(disable dpr-hack4)
(disable dpr-hack5)
(disable GCD-FACTORS-GIVES-LINEAR-COMBINATION-REWRITE)
(disable GCD-FACTORS-GIVES-LINEAR-COMBINATION)
(disable gcd-factors)
(enable idifference-iplus-canonicalizer1)
(disable ilessp)
(disable idifference)
(disable iplus)
(disable itimes)
(disable ineg)
(disable fix-int)
(disable izerop)
(disable iremainder)
(disable iquotient)


(prove-lemma gcd-remainder-times-fact1-proof
             (rewrite)
             (implies (equal (gcd a b) 1)
                      (equal (equal (remainder (times b c) a) 0)
                             (equal (remainder c a) 0)))
             )



;;;;;;;;;; we've proved our important gcd fact - let's get on with it!



(prove-lemma times-gcd-fact
             (rewrite)
             (implies (and (equal (gcd a c) 1)
                           (equal (gcd b d) 1))
                      (equal (equal (times a b) (times c d))
                             (and (equal (fix a) (fix d))
                                  (equal (fix b) (fix c)))))
             )
```

```
(prove-lemma equal-times-times-quotient-arg2 (rewrite)
             (implies
              (and
               (equal (remainder b e) 0)
               (equal (remainder d e) 0))
              (and
               (equal
                (equal (times a (quotient b e))
                       (times c (quotient d e)))
                (equal (times a b) (times c d)))
               (equal
                (equal (times (quotient b e) a)
                       (times c (quotient d e)))
                (equal (times a b) (times c d)))
               (equal
                (equal (times (quotient b e) a)
                       (times (quotient d e) c))
                (equal (times a b) (times c d))))))

(prove-lemma quotient-gcd-times-fact
             (rewrite)
             (implies (and (equal (times v z) (times w x))
                           (lessp 0 w)
                           (lessp 0 z))
                      (equal (quotient v (gcd v w))
                             (quotient x (gcd x z))))
             )


(lemma quotient-gcd-times-fact1 (rewrite)
       (implies (and (equal (times v z) (times w x))
                     (lessp 0 w)
                     (lessp 0 z))
                (equal (equal (quotient v (gcd v w))
                              (quotient x (gcd x z)))
                       t))
       ((enable quotient-gcd-times-fact)))

(lemma quotient-gcd-times-fact2 (rewrite)
       (implies (and (equal (times v z) (times w x))
                     (lessp 0 w)
                     (lessp 0 z))
                (equal (equal (quotient v (gcd w v))
                              (quotient x (gcd x z)))
                       t))
       ((enable quotient-gcd-times-fact commutativity-of-gcd)))

(lemma quotient-gcd-times-fact3 (rewrite)
       (implies (and (equal (times v z) (times w x))
                     (lessp 0 w)
                     (lessp 0 z))
                (equal (equal (quotient v (gcd v w))
                              (quotient x (gcd z x)))
                       t))
       ((enable quotient-gcd-times-fact commutativity-of-gcd)))

(lemma quotient-gcd-times-fact4 (rewrite)
       (implies (and (equal (times v z) (times w x))
                     (lessp 0 w)
                     (lessp 0 z))
                (equal (equal (quotient v (gcd w v))
                              (quotient x (gcd z x)))
                       t))
       ((enable quotient-gcd-times-fact commutativity-of-gcd)))
```

```
(prove-lemma quotient-gcd-times-fact5
             (rewrite)
             (implies (and (equal (times v z) (times w x))
                           (lessp 0 v)
                           (lessp 0 x))
                      (equal (equal (quotient v (gcd v w))
                                    (quotient x (gcd x z)))
                             t))
             )


(prove-lemma equal-times-gcd-bridge1 (rewrite)
             (implies (and (equal (quotient b (gcd c b))
                                  (quotient d (gcd a d)))
                           (equal (quotient c (gcd c b))
                                  (quotient a (gcd a d))))
                      (equal (equal (times a b) (times c d))
                             t))
             )



(prove-lemma numberp-if-integerp-and-not-negativep (rewrite)
             (implies
              (and
               (integerp x)
               (not (negativep x)))
              (numberp x))
             ((enable integerp)))

(prove-lemma negativep-if-integerp-and-not-numberp (rewrite)
             (implies
              (and
               (integerp x)
               (not (numberp x)))
              (negativep x))
             ((enable integerp)))

(disable numberp-if-integerp-and-not-negativep)
(disable negativep-if-integerp-and-not-numberp)



(prove-lemma requal-reduce-reduce-equal (rewrite)
             (equal
              (requal a b)
              (equal (reduce a) (reduce b)))
             ((enable numberp-if-integerp-and-not-negativep ineg)))

(disable requal-reduce-reduce-equal)




(prove-lemma commutativity-of-requal (rewrite)
             (equal (requal x y) (requal y x))
             ((disable fix-rational)
              (enable itimes)))
```

```
(prove-lemma requal-reduce1 (rewrite)
             (equal
              (requal (reduce x) y)
              (requal x y))
             ((enable integerp fix-int itimes)
              (disable itimes-negativep-arg1 itimes-negativep-arg2)))

(lemma requal-reduce2 (rewrite)
             (equal
              (requal x (reduce y))
              (requal x y))
             ((use (commutativity-of-requal (x x) (y (reduce y)))
                   (commutativity-of-requal))
              (enable requal-reduce1)
              (disable commutativity-of-requal)))

(prove-lemma reduce-reduce (rewrite)
             (equal (reduce (reduce x)) (reduce x))
             )

(prove-lemma rplus-open-up
 (rewrite)
 (equal (rplus a b)
        (if (rationalp a)
            (if (rationalp b)
                (reduce (rational (iplus (itimes (numerator a) (denominator b))
                                         (itimes (numerator b)
                                                 (denominator a)))
                                  (itimes (denominator a)
                                          (denominator b))))
                (reduce (fix-rational a)))
            (reduce (fix-rational b))))
 )

(disable rplus-open-up)

(lemma commutativity-of-rplus (rewrite)
       (equal (rplus x y)
              (rplus y x))
       ((enable rplus simple-rplus)
        (enable-theory arithmetic integers)))

(lemma rplus-requal-arg1 (rewrite)
       (implies
        (requal a b)
        (requal
         (rplus a x)
         (rplus b x)))
       ((enable itimes requal rplus-open-up requal-reduce1 requal-reduce2
                itimes-is-times rational-generalization fix-rational rationalp
                integerp integerp-minus fix-int-on-integers
                correctness-of-cancel-equal-times)
        (enable-theory integers arithmetic rational-defns)))

(disable rplus-requal-arg1)

(prove-lemma requal-x-x (rewrite)
             (requal x x))

(lemma rplus-reduce-arg1 (rewrite)
       (requal (rplus (reduce x) y)
               (rplus x y))
       ((enable rplus-requal-arg1 requal-reduce1 requal-x-x)))

(lemma rplus-reduce-arg2 (rewrite)
             (requal
              (rplus x (reduce y))
              (rplus x y))
             ((use (commutativity-of-rplus (x x) (y (reduce y)))
                   (commutativity-of-rplus (x x) (y y))
                   (rplus-reduce-arg1 (x y) (y x)))))

(lemma requal-simple-rplus-reduce-arg1 (rewrite)
       (requal
        (simple-rplus (reduce x) y)
        (simple-rplus x y))
       ((use (rplus-reduce-arg1))
        (enable rplus requal-reduce1 requal-reduce2)))

(lemma requal-simple-rplus-reduce-arg2 (rewrite)
       (requal
        (simple-rplus x (reduce y))
        (simple-rplus x y))
       ((use (rplus-reduce-arg2))
        (enable rplus requal-reduce1 requal-reduce2)))

(prove-lemma reduce-nrationalp (rewrite)
             (implies
              (not (rationalp x))
              (equal (reduce x) (rational 0 1))))

(prove-lemma numberp-numerator-reduce (rewrite)
             (equal
              (numberp (numerator (reduce x)))
              (numberp (numerator (fix-rational x)))))


(prove-lemma negativep-ineg (rewrite)
             (equal
              (negativep (ineg x))
              (lessp 0 x))
             ((enable ineg)))

(prove-lemma numberp-ineg (rewrite)
             (equal
              (numberp (ineg x))
              (not (lessp 0 x)))
             ((enable ineg)))

(prove-lemma rational-ineg-numerator-reduce-bridge (rewrite)
             (equal
              (rational (ineg (numerator (reduce x)))
                        (denominator (reduce x)))
              (reduce (rational (ineg (numerator x))
                                (denominator x))))
             ((enable ineg)))

(prove-lemma reduce-0 (rewrite)
             (equal
              (reduce (rational 0 x))
              (rational 0 1)))

(prove-lemma simple-rneg-reduce (rewrite)
             (equal
              (simple-rneg (reduce x))
              (reduce (simple-rneg x)))
             ((disable reduce)))
```

```
(lemma rneg-reduce (rewrite)
       (equal
        (rneg (reduce x))
        (rneg x))
       ((enable rneg simple-rneg-reduce reduce-reduce)))

(prove-lemma simple-rneg-simple-rneg (rewrite)
             (requal
              (simple-rneg (simple-rneg x))
              x))

(lemma requal-rneg-rneg (rewrite)
       (requal
        (rneg (rneg x))
        x)
       ((enable rneg rneg-reduce requal-reduce1 requal-reduce2
                simple-rneg-simple-rneg simple-rneg-reduce)))

(lemma rneg-rneg (rewrite)
       (equal
        (rneg (rneg x))
        (reduce x))
       ((use (requal-rneg-rneg))
        (enable requal-reduce-reduce-equal rneg reduce-reduce)))

(prove-lemma rationalp-means (rewrite)
             (implies
              (rationalp x)
              (and
               (rational-formp x)
               (integerp (numerator x))
               (lessp 0 (denominator x)))))

(prove-lemma means-rationalp
             (rewrite)
             (implies (and (integerp n) (lessp 0 d))
                      (rationalp (rational n d))))

(prove-lemma nrational-rplus-arg1 (rewrite)
             (implies
              (not (rationalp x))
              (equal
               (rplus x y)
               (reduce y)))
             ((disable rationalp)))

(lemma nrational-rplus-arg2 (rewrite)
       (implies
        (not (rationalp x))
        (equal
         (rplus y x)
         (reduce y)))
       ((enable commutativity-of-rplus nrational-rplus-arg1)))


(prove-lemma nrational-simple-rplus-arg1 (rewrite)
             (implies
              (not (rationalp x))
              (equal
               (simple-rplus x y)
               (fix-rational y)))
             ((disable rationalp)))
```

```
(prove-lemma nrational-simple-rplus-arg2 (rewrite)
             (implies
              (not (rationalp x))
              (equal
               (simple-rplus y x)
               (fix-rational y)))
             ((disable rationalp)))

(lemma simple-rplus-fix-rational-arg1 (rewrite)
       (equal
        (simple-rplus (fix-rational x) y)
        (simple-rplus x y))
       ((enable simple-rplus fix-rational-fix-rational)))

(lemma simple-rplus-fix-rational-arg2 (rewrite)
       (equal
        (simple-rplus x (fix-rational y))
        (simple-rplus x y))
       ((enable simple-rplus fix-rational-fix-rational)))

(prove-lemma fix-rational-of-rationalp (rewrite)
             (implies
              (rationalp x)
              (equal (fix-rational x) x)))

(prove-lemma negative-guts-ineg (rewrite)
             (equal
              (negative-guts (ineg x))
              (if (lessp 0 x)
                  x
                0))
             ((enable ineg)))

(prove-lemma rationalp-simple-rplus
             (rewrite)
             (rationalp (simple-rplus x y))
             )

(prove-lemma fix-rational-simple-rplus (rewrite)
             (equal
              (fix-rational (simple-rplus x y))
              (simple-rplus x y))
             ((disable simple-rplus)))

;;needlessly complex proof
(prove-lemma requal-associativity-of-simple-rplus
             (rewrite)
             (requal (simple-rplus (simple-rplus x y) z)
                     (simple-rplus x (simple-rplus y z)))
             )



;;;  equal-times bridge lemmas

(prove-lemma equal-times-bridge1
             (rewrite)
             (implies (and (equal (times a b) (times c d))
                           (equal (times a x) (times c y))
                           (not (zerop a)))
                      (equal (equal (times b y) (times d x))
                             t))
             )
```

```
(lemma equal-times-bridge2 (rewrite)
       (implies (and (equal (times a b) (times c d))
                     (equal (times a x) (times c y))
                     (not (zerop a)))
                (equal (equal (times y b) (times d x)) t))
       ((enable equal-times-bridge1 commutativity-of-times)))

(lemma equal-times-bridge3 (rewrite)
       (implies (and (equal (times a b) (times c d))
                     (equal (times a x) (times c y))
                     (not (zerop a)))
                (equal (equal (times b y) (times x d)) t))
       ((enable equal-times-bridge1 commutativity-of-times)))

(lemma equal-times-bridge4 (rewrite)
       (implies (and (equal (times a b) (times c d))
                     (equal (times a x) (times c y))
                     (not (zerop a)))
                (equal (equal (times y b) (times x d)) t))
       ((enable equal-times-bridge1 commutativity-of-times)))

(prove-lemma transitivity-of-requal-bridge (rewrite)
             (implies
              (and
               (requal a b)
               (requal b c))
              (requal a c))
             ((enable itimes integerp ineg)
              (disable rationalp)))

(disable transitivity-of-requal-bridge)

(lemma transitivity-of-requal (rewrite)
       (and
        (implies
         (and
          (requal a b)
          (requal b c))
         (requal a c))
        (implies
         (and
          (requal a b)
          (requal c b))
         (requal a c))
        (implies
         (and
          (requal b a)
          (requal b c))
         (requal a c))
        (implies
         (and
          (requal b a)
          (requal c b))
         (requal a c)))
       ((enable transitivity-of-requal-bridge commutativity-of-requal)))

(prove-lemma equal-requal-rewrite
             (rewrite)
             (and (implies (requal b c)
                           (equal (requal a b) (requal a c)))
                  (implies (requal b c)
                           (equal (requal a b) (requal c a)))
                  (implies (requal b c)
                           (equal (requal b a) (requal c a))))
             )

(disable equal-requal-rewrite)

(lemma requal-simple-rplus-bridge (rewrite)
        (and
         (equal
          (requal (simple-rplus (reduce x) y) z)
          (requal (simple-rplus x y) z))
         (equal
          (requal (simple-rplus x (reduce y)) z)
          (requal (simple-rplus x y) z))
         (equal
          (requal z (simple-rplus (reduce x) y))
          (requal z (simple-rplus x y)))
         (equal
          (requal z (simple-rplus x (reduce y)))
          (requal z (simple-rplus x y))))
        ((use (requal-simple-rplus-reduce-arg1)
              (requal-simple-rplus-reduce-arg2))
         (enable equal-requal-rewrite transitivity-of-requal
                 commutativity-of-requal)))

(lemma requal-associativity-of-rplus (rewrite)
       (requal (rplus (rplus x y) z)
               (rplus x (rplus y z)))
       ((enable requal-associativity-of-simple-rplus rplus
                requal-simple-rplus-bridge
                requal-reduce1 requal-reduce2)))

(lemma associativity-of-rplus (rewrite)
       (equal
        (rplus (rplus x y) z)
        (rplus x (rplus y z)))
       ((use (requal-associativity-of-rplus))
        (enable requal-reduce-reduce-equal rplus reduce-reduce )))

(lemma commutativity2-of-rplus (rewrite)
       (equal (rplus x (rplus y z))
              (rplus y (rplus x z)))
       ((use (associativity-of-rplus (x x) (y z) (z y)))
        (enable commutativity-of-rplus)))


(lemma rplus-rdifference-arg1 (rewrite)
       (equal
        (rplus (rdifference x y) z)
        (rdifference (rplus x z) y))
       ((enable rdifference commutativity-of-rplus commutativity2-of-rplus)))

(lemma rplus-rdifference-arg2 (rewrite)
       (equal
        (rplus x (rdifference y z))
        (rdifference (rplus x y) z))
       ((enable rdifference commutativity-of-rplus commutativity2-of-rplus)))


(lemma reduce-rplus (rewrite)
       (equal
        (reduce (rplus x y))
        (rplus x y))
       ((enable rplus reduce-reduce)))

(lemma reduce-rtimes (rewrite)
       (equal
        (reduce (rtimes x y))
        (rtimes x y))
       ((enable rtimes reduce-reduce)))
```

```
(lemma reduce-difference (rewrite)
       (equal
        (reduce (rdifference x y))
        (rdifference x y))
       ((enable rdifference reduce-rplus)))

(lemma reduce-rquotient (rewrite)
       (equal
        (reduce (rquotient x y))
        (rquotient x y))
       ((enable rquotient reduce-rtimes)))

(lemma reduce-rmagnitude (rewrite)
       (equal
        (reduce (rmagnitude x))
        (rmagnitude x))
       ((enable rmagnitude reduce-reduce)))

(lemma reduce-rneg (rewrite)
       (equal
        (reduce (rneg x))
        (rneg x))
       ((enable rneg reduce-reduce)))

(lemma rplus-reduce-arg1-rewrite (rewrite)
       (equal
        (rplus (reduce x) y)
        (rplus x y))
       ((use (rplus-reduce-arg1))
        (enable requal-reduce-reduce-equal reduce-rplus)))

(lemma rplus-reduce-arg2-rewrite (rewrite)
       (equal
        (rplus x (reduce y))
        (rplus x y))
       ((use (rplus-reduce-arg2))
        (enable requal-reduce-reduce-equal reduce-rplus)))

(prove-lemma requal-rneg-simple-rplus
             (rewrite)
             (requal (simple-rplus (simple-rneg x)
                                   (simple-rneg y))
                     (simple-rneg (simple-rplus x y)))
             )

(prove-lemma requal-rplus-rneg
             (rewrite)
             (requal (rneg (rplus x y))
                     (rplus (rneg x) (rneg y)))
             )

(lemma rneg-rplus (rewrite)
       (equal (rneg (rplus x y))
              (rplus (rneg x) (rneg y)))
       ((use (requal-rplus-rneg))
        (enable requal-reduce-reduce-equal reduce-rneg reduce-rplus)))


(lemma rdifference-rdifference-arg1 (rewrite)
       (equal
        (rdifference (rdifference x y) z)
        (rdifference x (rplus y z)))
       ((enable rdifference commutativity-of-rplus commutativity2-of-rplus
                rneg-rplus)))

(lemma rdifference-rdifference-arg2 (rewrite)
       (equal
        (rdifference x (rdifference y z))
        (rdifference (rplus x z) y))
       ((enable rdifference commutativity-of-rplus commutativity2-of-rplus
                rneg-rplus rneg-rneg rplus-reduce-arg1-rewrite)))

(lemma rplus-rneg-arg1 (rewrite)
       (equal
        (rplus (rneg x) y)
        (rdifference y x))
       ((enable rdifference commutativity-of-rplus)))

(lemma rplus-rneg-arg2 (rewrite)
       (equal
        (rplus x (rneg y))
        (rdifference x y))
       ((enable rdifference commutativity-of-rplus)))


;;; times, quotient


(prove-lemma commutativity-of-simple-rtimes (rewrite)
             (equal
              (simple-rtimes x y)
              (simple-rtimes y x)))

(lemma commutativity-of-rtimes (rewrite)
       (equal
        (rtimes x y)
        (rtimes y x))
       ((enable commutativity-of-simple-rtimes rtimes)))


;;;; appears to work very slowly - speed it up with concept of rzerop
;(lemma associativity-of-simple-rtimes (rewrite)
;       (requal
;       (simple-rtimes (simple-rtimes x y) z)
;       (simple-rtimes x (simple-rtimes y z)))
;       ((enable-theory arithmetic rational-defns)
;       (enable simple-rtimes itimes requal fix-rational fix-int integerp)))

(prove-lemma simple-rtimes-rzerop (rewrite)
             (implies
              (rzerop x)
              (and
               (equal (numerator (simple-rtimes x y)) 0)
               (equal (numerator (simple-rtimes y x)) 0))))


(lemma associativity-of-simple-rtimes-when-not-rzerop (rewrite)
       (implies
        (and
         (not (rzerop x))
         (not (rzerop y))
         (not (rzerop z)))
        (requal
         (simple-rtimes (simple-rtimes x y) z)
         (simple-rtimes x (simple-rtimes y z))))
        ((enable-theory arithmetic rational-defns)
        (enable simple-rtimes itimes requal rationalp
                fix-int integerp rzerop fix-rational)))
```

```
(prove-lemma numerator-zero-rzerop-bridge (rewrite)
             (implies
              (equal (numerator x) 0)
              (rzerop x)))

(lemma associativity-of-simple-rtimes (rewrite)
        (requal
         (simple-rtimes (simple-rtimes x y) z)
         (simple-rtimes x (simple-rtimes y z)))
        ((use (associativity-of-simple-rtimes-when-not-rzerop))
         (enable requal simple-rtimes-rzerop numerator-zero-rzerop-bridge
                 fix-rational itimes)
         (enable-theory rational-defns)))

(lemma requal-simple-rtimes-requal-arg1 (rewrite)
       (implies
        (requal x y)
        (requal
         (simple-rtimes x z)
         (simple-rtimes y z)))
       ((enable requal simple-rtimes itimes fix-rational fix-int integerp
                rationalp-means means-rationalp
                correctness-of-cancel-equal-times)
        (enable-theory arithmetic rational-defns)))

(lemma requal-simple-rtimes-requal-arg2 (rewrite)
       (implies
        (requal x y)
        (requal
         (simple-rtimes z x)
         (simple-rtimes z y)))
       ((enable requal-simple-rtimes-requal-arg1
                commutativity-of-simple-rtimes)))

(lemma requal-simple-rtimes-reduce-arg1 (rewrite)
       (requal
        (simple-rtimes (reduce x) y)
        (simple-rtimes x y))
       ((enable requal-simple-rtimes-requal-arg1 requal-x-x requal-reduce1)))

(lemma requal-simple-rtimes-reduce-arg2 (rewrite)
       (requal
        (simple-rtimes x (reduce y))
        (simple-rtimes x y))
       ((enable requal-simple-rtimes-requal-arg2 requal-x-x requal-reduce1)))

(lemma requal-simple-rtimes-bridge (rewrite)
       (and
        (equal
         (requal (simple-rtimes (reduce x) y) z)
         (requal (simple-rtimes x y) z))
        (equal
         (requal (simple-rtimes x (reduce y)) z)
         (requal (simple-rtimes x y) z))
        (equal
         (requal z (simple-rtimes (reduce x) y))
         (requal z (simple-rtimes x y)))
        (equal
         (requal z (simple-rtimes x (reduce y)))
         (requal z (simple-rtimes x y))))
       ((use (requal-simple-rtimes-reduce-arg1)
             (requal-simple-rtimes-reduce-arg2))
        (enable equal-requal-rewrite transitivity-of-requal
                commutativity-of-requal)))
```

```
(lemma requal-associativity-of-rtimes (rewrite)
       (requal
        (rtimes (rtimes x y) z)
        (rtimes x (rtimes y z)))
       ((enable rtimes requal-simple-rtimes-bridge
                associativity-of-simple-rtimes requal-reduce1 requal-reduce2)))

(lemma associativity-of-rtimes (rewrite)
       (equal
        (rtimes (rtimes x y) z)
        (rtimes x (rtimes y z)))
       ((use (requal-associativity-of-rtimes))
        (enable reduce-rtimes requal-reduce-reduce-equal)))


(prove-lemma simple-rplus-rzerop (rewrite)
             (implies
              (rzerop x)
             (and
              (requal (simple-rplus x y) y)
              (requal (simple-rplus y x) y))))

(prove-lemma numerator-type (rewrite)
             (or
              (numberp (numerator x))
              (negativep (numerator x))))

(prove-lemma rationalp-simple-rtimes (rewrite)
             (rationalp (simple-rtimes x y)))


(prove-lemma fix-rational-simple-rtimes (rewrite)
             (equal
              (fix-rational (simple-rtimes x y))
              (simple-rtimes x y))
             ((disable rationalp)))

(prove-lemma requal-simple-rtimes-simple-rplus-when-not-rzerop
             (rewrite)
             (implies (and (not (rzerop x))
                           (not (rzerop y))
                           (not (rzerop z)))
                      (requal (simple-rtimes (simple-rplus x y) z)
                              (simple-rplus (simple-rtimes x z)
                                            (simple-rtimes y z))))
             )

(prove-lemma requal-rzerop-simple-rplus
             (rewrite)
             (implies (rzerop x)
                      (and (requal (simple-rplus x y)
                                   (fix-rational y))
                           (requal (simple-rplus y x)
                                   (fix-rational y))))
             )

(prove-lemma commutativity-of-simple-rplus (rewrite)
             (equal
              (simple-rplus x y)
              (simple-rplus y x)))
```

```
(prove-lemma simple-rplus-bridge
  (rewrite)
  (implies (or (rzerop x) (rzerop y) (rzerop z))
           (requal (simple-rtimes (simple-rplus x y) z)
                   (simple-rplus (simple-rtimes x z)
                                 (simple-rtimes y z))))
  )

(lemma requal-simple-rtimes-simple-rplus-arg1 (rewrite)
       (requal
        (simple-rtimes (simple-rplus x y) z)
        (simple-rplus (simple-rtimes x z) (simple-rtimes y z)))
       ((use (requal-simple-rtimes-simple-rplus-when-not-rzerop))
        (enable simple-rplus-bridge)))


(lemma requal-rtimes-rplus-bridge (rewrite)
       (requal
        (rtimes (rplus x y) z)
        (rplus (rtimes x z) (rtimes y z)))
       ((enable rtimes rplus requal-simple-rtimes-bridge
                requal-simple-rplus-bridge requal-reduce1 requal-reduce2
                requal-simple-rtimes-simple-rplus-arg1)))

(lemma rtimes-rplus-arg1 (rewrite)
       (equal
        (rtimes (rplus x y) z)
        (rplus (rtimes x z) (rtimes y z)))
       ((use (requal-rtimes-rplus-bridge))
        (enable  requal-reduce-reduce-equal reduce-rplus reduce-rtimes)))

(lemma rtimes-rplus-arg2 (rewrite)
       (equal
        (rtimes x (rplus y z) )
        (rplus (rtimes x y) (rtimes x z)))
       ((use (rtimes-rplus-arg1 (x z) (z x)))
        (enable commutativity-of-rtimes commutativity-of-rplus)))


(prove-lemma requal-simple-rtimes-simple-rneg (rewrite)
             (requal
              (simple-rtimes (simple-rneg x) y)
              (simple-rneg (simple-rtimes x y)))
             ((enable ineg izerop) (disable rationalp)))


(lemma requal-rtimes-rneg (rewrite)
       (requal
        (rtimes (rneg x) y)
        (rneg (rtimes x y)))
       ((enable rtimes rneg requal-simple-rtimes-bridge simple-rneg-reduce
                requal-reduce1 requal-reduce2
                requal-simple-rtimes-simple-rneg)))

(lemma rtimes-rneg-arg1 (rewrite)
       (equal
        (rtimes (rneg x) y)
        (rneg (rtimes x y)))
       ((use (requal-rtimes-rneg))
        (enable requal-reduce-reduce-equal reduce-rtimes reduce-rneg)))

(lemma rtimes-rneg-arg2 (rewrite)
       (equal
        (rtimes x (rneg y))
        (rneg (rtimes x y)))
       ((use (rtimes-rneg-arg1 (x y) (y x)))
        (enable commutativity-of-rtimes)))
```

```
(lemma rtimes-rdifference-arg1 (rewrite)
       (equal
        (rtimes (rdifference x y) z)
        (rdifference (rtimes x z) (rtimes y z)))
       ((enable rdifference rtimes-rplus-arg1 rtimes-rneg-arg1)))

(lemma rtimes-rdifference-arg2 (rewrite)
       (equal
        (rtimes x (rdifference y z))
        (rdifference (rtimes x y) (rtimes x z)))
       ((enable rdifference rtimes-rplus-arg2 rtimes-rneg-arg2)))

(prove-lemma rneg-rdifference
             (rewrite)
             (equal (rneg (rdifference x y))
                    (rdifference y x))
             )

(prove-lemma rdifference-rneg-arg2 (rewrite)
             (equal (rdifference x (rneg y))
                    (rplus x y))
             ((disable rplus rneg)))

(disable rdifference-rneg-arg2)
(disable rneg-rdifference)
(disable RTIMES-RDIFFERENCE-ARG2)
(disable RTIMES-RDIFFERENCE-ARG1)
(disable RTIMES-RNEG-ARG2)
(disable RTIMES-RNEG-ARG1)
(disable REQUAL-RTIMES-RNEG)
(disable REQUAL-SIMPLE-RTIMES-SIMPLE-RNEG)
(disable RTIMES-RPLUS-ARG2)
(disable RTIMES-RPLUS-ARG1)
(disable REQUAL-RTIMES-RPLUS-BRIDGE)
(disable REQUAL-SIMPLE-RTIMES-SIMPLE-RPLUS-ARG1)
(disable SIMPLE-RPLUS-BRIDGE)
(disable COMMUTATIVITY-OF-SIMPLE-RPLUS)
(disable REQUAL-RZEROP-SIMPLE-RPLUS)
(disable REQUAL-SIMPLE-RTIMES-SIMPLE-RPLUS-WHEN-NOT-RZEROP)
(disable FIX-RATIONAL-SIMPLE-RTIMES)
(disable RATIONALP-SIMPLE-RTIMES)
(disable NUMERATOR-TYPE)
(disable SIMPLE-RPLUS-RZEROP)
(disable ASSOCIATIVITY-OF-RTIMES)
(disable REQUAL-ASSOCIATIVITY-OF-RTIMES)
(disable REQUAL-SIMPLE-RTIMES-BRIDGE)
(disable REQUAL-SIMPLE-RTIMES-REDUCE-ARG2)
(disable REQUAL-SIMPLE-RTIMES-REDUCE-ARG1)
(disable REQUAL-SIMPLE-RTIMES-REQUAL-ARG2)
(disable REQUAL-SIMPLE-RTIMES-REQUAL-ARG1)
(disable ASSOCIATIVITY-OF-SIMPLE-RTIMES)
(disable NUMERATOR-ZERO-RZEROP-BRIDGE)
(disable ASSOCIATIVITY-OF-SIMPLE-RTIMES-WHEN-NOT-RZEROP)
(disable SIMPLE-RTIMES-RZEROP)
(disable RZEROP)
(disable COMMUTATIVITY-OF-RTIMES)
(disable COMMUTATIVITY-OF-SIMPLE-RTIMES)
(disable RPLUS-RNEG-ARG2)
(disable RPLUS-RNEG-ARG1)
(disable RDIFFERENCE-RDIFFERENCE-ARG2)
(disable RDIFFERENCE-RDIFFERENCE-ARG1)
(disable RNEG-RPLUS)
(disable REQUAL-RPLUS-RNEG)
(disable REQUAL-RNEG-SIMPLE-RPLUS)
(disable RPLUS-REDUCE-ARG2-REWRITE)
(disable RPLUS-REDUCE-ARG1-REWRITE)
(disable REDUCE-RNEG)
(disable REDUCE-RMAGNITUDE)
```

```
(disable REDUCE-RQUOTIENT)
(disable REDUCE-DIFFERENCE)
(disable REDUCE-RTIMES)
(disable REDUCE-RPLUS)
(disable RPLUS-RDIFFERENCE-ARG2)
(disable RPLUS-RDIFFERENCE-ARG1)
(disable COMMUTATIVITY2-OF-RPLUS)
(disable ASSOCIATIVITY-OF-RPLUS)
(disable REQUAL-ASSOCIATIVITY-OF-RPLUS)
(disable REQUAL-SIMPLE-RPLUS-BRIDGE)
(disable EQUAL-REQUAL-REWRITE)
(disable TRANSITIVITY-OF-REQUAL)
(disable TRANSITIVITY-OF-REQUAL-BRIDGE)
(disable EQUAL-TIMES-BRIDGE4)
(disable EQUAL-TIMES-BRIDGE3)
(disable EQUAL-TIMES-BRIDGE2)
(disable EQUAL-TIMES-BRIDGE1)
(disable REQUAL-ASSOCIATIVITY-OF-SIMPLE-RPLUS)
(disable FIX-RATIONAL-SIMPLE-RPLUS)
(disable RATIONALP-SIMPLE-RPLUS)
(disable NEGATIVE-GUTS-INEG)
(disable FIX-RATIONAL-OF-RATIONALP)
(disable SIMPLE-RPLUS-FIX-RATIONAL-ARG2)
(disable SIMPLE-RPLUS-FIX-RATIONAL-ARG1)
(disable NRATIONAL-SIMPLE-RPLUS-ARG2)
(disable NRATIONAL-SIMPLE-RPLUS-ARG1)
(disable NRATIONAL-RPLUS-ARG2)
(disable NRATIONAL-RPLUS-ARG1)
(disable MEANS-RATIONALP)
(disable RATIONALP-MEANS)
(disable RNEG-RNEG)
(disable REQUAL-RNEG-RNEG)
(disable SIMPLE-RNEG-SIMPLE-RNEG)
(disable RNEG-REDUCE)
(disable SIMPLE-RNEG-REDUCE)
(disable REDUCE-0)
(disable RATIONAL-INEG-NUMERATOR-REDUCE-BRIDGE)
(disable NUMBERP-INEG)
(disable NEGATIVEP-INEG)
(disable NUMBERP-NUMERATOR-REDUCE)
(disable REDUCE-NRATIONALP)
(disable REQUAL-SIMPLE-RPLUS-REDUCE-ARG2)
(disable REQUAL-SIMPLE-RPLUS-REDUCE-ARG1)
(disable RPLUS-REDUCE-ARG2)
(disable RPLUS-REDUCE-ARG1)
(disable REQUAL-X-X)
(disable RPLUS-REQUAL-ARG1)
(disable COMMUTATIVITY-OF-RPLUS)
(disable RPLUS-OPEN-UP)
(disable REDUCE-REDUCE)
(disable REQUAL-REDUCE2)
(disable REQUAL-REDUCE1)
(disable COMMUTATIVITY-OF-REQUAL)
(disable REQUAL-REDUCE-REDUCE-EQUAL)
(disable NEGATIVEP-IF-INTEGERP-AND-NOT-NUMBERP)
(disable NUMBERP-IF-INTEGERP-AND-NOT-NEGATIVEP)
(disable EQUAL-TIMES-GCD-BRIDGE1)
(disable QUOTIENT-GCD-TIMES-FACT5)
(disable QUOTIENT-GCD-TIMES-FACT4)
(disable QUOTIENT-GCD-TIMES-FACT3)
(disable QUOTIENT-GCD-TIMES-FACT2)
(disable QUOTIENT-GCD-TIMES-FACT1)
(disable QUOTIENT-GCD-TIMES-FACT)
(disable EQUAL-TIMES-TIMES-QUOTIENT-ARG2)
(disable TIMES-GCD-FACT)
(disable GCD-REMAINDER-TIMES-FACT1-PROOF)
(disable DIVIDES-PRODUCT-REDUCTION)
(disable REMAINDER-0-SUFFICIENCY)
```

```
(disable DPR-HACK5)
(disable DPR-HACK4)
(disable DPR-HACK3)
(disable DPR-HACK2)
(disable DPR-HACK1)
(disable GCD-FACTORS-GIVES-LINEAR-COMBINATION-REWRITE)
(disable GCD-FACTORS-GIVES-LINEAR-COMBINATION)
(disable GCD-FACTORS)
(disable DIVIDES-EACH-EQUALITY)
(disable GCD-QUOTIENT-QUOTIENT)
(disable GCD-TIMES2)
(disable GCD-TIMES1)
(disable GCD-TIMES1-INDUCT)
(disable GCD-REMAINDER-FACT2)
(disable GCD-REMAINDER-FACT1)
(disable RATIONAL-GENERALIZATION)
(disable FIX-RATIONAL-RMAGNITUDE)
(disable FIX-RATIONAL-RQUOTIENT)
(disable FIX-RATIONAL-RDIFFERENCE)
(disable FIX-RATIONAL-RNEG)
(disable FIX-RATIONAL-RTIMES)
(disable FIX-RATIONAL-FIX-RATIONAL)
(disable FIX-RATIONAL-RPLUS)
(disable FIX-RATIONAL-REDUCE)
(disable RATIONALP-RMAGNITUDE)
(disable RATIONALP-RQUOTIENT)
(disable RATIONALP-RDIFFERENCE)
(disable RATIONALP-RNEG)
(disable RATIONALP-RTIMES)
(disable RATIONALP-FIX-RATIONAL)
(disable RATIONALP-RPLUS)
(disable RATIONALP-REDUCE)
(disable IPLUS-IS-PLUS)
(disable ITIMES-IS-TIMES)
(disable EQUAL-INEG-INEG)
(disable ITIMES-NEGATIVEP-ARG2)
(disable ITIMES-NEGATIVEP-ARG1)
(disable INTEGERP-IF-NUMBERP)
(disable INTEGERP-IF-NEGATIVEP-NON-ZERO)
(disable ITIMES-INEG-ARG2)
(disable ITIMES-INEG-ARG1)
(disable FIX-INT-ON-INTEGERS)
(disable INTEGERP-MINUS)
(disable REQUAL)
(disable RLESSP)
(disable RMAGNITUDE)
(disable SIMPLE-RMAGNITUDE)
(disable RQUOTIENT)
(disable SIMPLE-RQUOTIENT)
(disable RTIMES)
(disable SIMPLE-RTIMES)
(disable RDIFFERENCE)
(disable RNEG)
(disable SIMPLE-RNEG)
(disable RPLUS)
(disable SIMPLE-RPLUS)
(disable REDUCE)
(disable FIX-RATIONAL)
(disable RATIONALP)
```

```
(deftheory r1
  (
   rdifference-rneg-arg2 rneg-rdifference rtimes-rdifference-arg2
   rtimes-rdifference-arg1 rtimes-rneg-arg2 rtimes-rneg-arg1 rtimes-rplus-arg2
   rtimes-rplus-arg1  associativity-of-rtimes
    rzerop commutativity-of-rtimes
    rplus-rneg-arg2 rplus-rneg-arg1 rdifference-rdifference-arg2
    rdifference-rdifference-arg1 rneg-rplus
    rplus-reduce-arg2-rewrite rplus-reduce-arg1-rewrite
    reduce-rneg reduce-rmagnitude reduce-rquotient reduce-difference
    reduce-rtimes reduce-rplus rplus-rdifference-arg2 rplus-rdifference-arg1
    commutativity2-of-rplus associativity-of-rplus
    equal-requal-rewrite transitivity-of-requal  fix-rational-of-rationalp
    nrational-rplus-arg2  nrational-rplus-arg1  rationalp-means means-rationalp
    rneg-rneg rneg-reduce reduce-0  numberp-numerator-reduce  reduce-nrationalp
    rplus-reduce-arg2   rplus-reduce-arg1   requal-x-x   rplus-requal-arg1
    commutativity-of-rplus   reduce-reduce   requal-reduce2   requal-reduce1
    commutativity-of-requal   rational-generalization   fix-rational-rmagnitude
    fix-rational-rquotient   fix-rational-rdifference   fix-rational-rneg
    fix-rational-rtimes   fix-rational-fix-rational   fix-rational-rplus
    fix-rational-reduce   rationalp-rmagnitude   rationalp-rquotient
    rationalp-rdifference   rationalp-rneg   rationalp-rtimes
    rationalp-fix-rational   rationalp-rplus   rationalp-reduce ))


(definition rplus-tree (x)
  (if (nlistp x) (list 'rational ''0 ''1)
    (if (nlistp (cdr x)) (list 'reduce (car x))
        (if (nlistp (cddr x)) (list 'rplus (car x) (cadr x))
          (list 'rplus (car x) (rplus-tree (cdr x)))))))

#|
(definition rplus-tree (x)
  (if (nlistp x) '(rational 0 1)
    (if (nlistp (cdr x)) (list 'reduce (car x))
        (if (nlistp (cddr x)) (list 'rplus (car x) (cadr x))
          (list 'rplus (car x) (rplus-tree (cdr x)))))))
|#

(definition rplus-fringe (x)
  (if (and (listp x) (equal (car x) 'rplus))
      (append (rplus-fringe (cadr x)) (rplus-fringe (caddr x)))
    (cons x nil)))


(definition split-by-parity (x)
  (if (listp x)
      (let ((rest (split-by-parity (cdr x))))
        (if (and (equal (caar x) 'rneg) (not (equal (caadar x) 'rneg) )
                 (equal (cddar x) nil))
            (cons (car rest) (cons (cadar x) (cdr rest)))
          (cons (cons (car x) (car rest)) (cdr rest))))
    (cons nil nil)))

(definition make-negs (x)
  (if (listp x)
      (cons (list 'rneg (car x))
            (make-negs (cdr x)))
    nil))

#| old definition changed 8/29
(definition cancel-rplus (x)
  (if (equal (car x) 'rplus)
      (let ((addends (rplus-fringe x)))
        (let ((pos (car (split-by-parity addends)))
              (neg (cdr (split-by-parity addends))))
          (let ((cancelled (bagint pos neg)))
            (if cancelled
                (rplus-tree (append (bagdiff pos cancelled)
                                    (make-negs (bagdiff neg cancelled))))
              x))))
    x))
|#

(definition cancel-rplus (x)
  (if (equal (car x) 'rplus)
      (let ((addends (rplus-fringe x)))
        (let ((pos (car (split-by-parity addends)))
              (neg (cdr (split-by-parity addends))))
          (let ((cancelled (bagint pos neg)))
            (if (listp cancelled)
                (rplus-tree (append (bagdiff pos cancelled)
                                    (make-negs (bagdiff neg cancelled))))
              x))))
    x))


(prove-lemma rplus-rzerop-bridge (rewrite)
             (implies
              (and
               (not (zerop v))
               (numberp z))
              (equal (reduce (rational (times v z) (times v w)))
                     (reduce (rational z w))))
             ((enable reduce rationalp)
              (enable-theory r1)))

(prove-lemma rplus-rzerop-bridge2 (rewrite)
             (implies
              (not (zerop v))
              (equal (reduce (rational (minus (times d v))
                                       (times v w)))
                     (reduce (rational (minus d) w))))
             ((enable reduce rationalp)
              (enable-theory r1)))

(prove-lemma rplus-rzerop (rewrite)
             (implies
              (rzerop x)
              (and
               (equal (rplus x y) (reduce y))
               (equal (rplus y x) (reduce y))))
             ((enable rplus simple-rplus fix-int-on-integers iplus itimes
                      fix-rational)
              (enable-theory r1)))

(prove-lemma eval$-rplus (rewrite)
             (implies
              (equal (car x) 'rplus)
             (equal
              (eval$ t x a)
              (rplus (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))
```

```
(prove-lemma eval$-reduce (rewrite)
             (implies
              (equal (car x) 'reduce)
             (equal
              (eval$ t x a)
              (reduce (eval$ t (cadr x) a)))))

(prove-lemma reduce-eval$-rplus-tree (rewrite)
             (equal (reduce (eval$ t (rplus-tree y) a))
                    (eval$ t (rplus-tree y) a))
             ((enable-theory r1)))

(prove-lemma rplus-eval$-rplus-tree (rewrite)
             (equal
              (eval$ t (rplus-tree (append x y)) a)
              (rplus (eval$ t (rplus-tree x) a) (eval$ t (rplus-tree y) a)))
             ((enable-theory r1)))


(prove-lemma member-append (rewrite)
             (equal
              (member a (append x y))
              (or
               (member a x)
               (member a y))))


(prove-lemma delete-append (rewrite)
             (equal
              (delete x (append a b))
              (if (member x a)
                  (append (delete x a) b)
                (append a (delete x b)))))


(prove-lemma member-subbagp (rewrite)
             (implies
              (and
               (member a x)
               (subbagp x y))
              (member a y)))


;;;; next 10 or so events done with Matt K.

(defn badguy (x y)
  (if (listp x)
      (if (member (car x) y)
          (badguy (cdr x) (delete (car x) y))
        (car x))
    0))


(prove-lemma member-occur (rewrite)
  (equal (member a x)
         (lessp 0 (occurrences a x))))


; simpler than in library
```

```
(prove-lemma occurrences-delete2 (rewrite)
  (equal (occurrences a (delete b x))
         (if (equal a b)
             (sub1 (occurrences a x))
           (occurrences a x)))
  ((disable-theory sets-and-bags)))


(prove-lemma subbagp-wit-lemma (rewrite)
  (equal (subbagp x y)
         (not (lessp (occurrences (badguy x y) y)
                     (occurrences (badguy x y) x))))
  ((disable-theory sets-and-bags)))


(prove-lemma occurrences-append (rewrite)
  (equal (occurrences a (append x y))
         (plus (occurrences a x) (occurrences a y)))
  ((disable-theory  sets-and-bags)))


(prove-lemma subbagp-append (rewrite)
             (subbagp (append x y) (append y x))
             ((disable-theory  sets-and-bags)))

(disable member-occur) (disable subbagp-wit-lemma)


(prove-lemma subbagp-delete-same (rewrite)
             (implies
              (subbagp x y)
              (subbagp (delete a x) (delete a y))))

(prove-lemma subbagp-delete-same-means (rewrite)
             (implies
              (and
               (member a y)
               (subbagp (delete a x) (delete a y)))
              (subbagp x y)))

(prove-lemma subbagp-delete-car (rewrite)
             (implies
              (subbagp x y)
              (subbagp (delete (car y) x) (cdr y)))
             ((use (subbagp-delete-same (x x) (y y) (a (car y))))))


(prove-lemma subbagp-delete-car2 (rewrite)
             (implies
              (subbagp x y)
              (subbagp (cdr x) (delete (car x) y))))


(prove-lemma subbagp-permutation (rewrite)
             (implies
              (and
               (subbagp x y)
               (subbagp y x))
              (permutation x y)))


(prove-lemma permutation-a-b (rewrite)
             (permutation (append a b) (append b a)))
```

```
(prove-lemma not-subbagp-not-permutation (rewrite)
             (implies
              (not (subbagp x y))
              (and
               (not (permutation x y))
               (not (permutation y x)))))

(prove-lemma permutation-as-subbagp-helper (rewrite)
             (iff
              (permutation x y)
              (and
               (subbagp x y)
               (subbagp y x))))

(prove-lemma permutation-as-subbagp (rewrite)
             (equal
              (permutation x y)
              (and
               (subbagp x y)
               (subbagp y x)))
             ((use (permutation-as-subbagp-helper))))


(disable permutation-as-subbagp-helper)
(disable permutation-as-subbagp)


(prove-lemma subbagp-necc (rewrite)
  (implies (subbagp x y)
           (not (lessp (occurrences a y) (occurrences a x))))
  ((enable subbagp-wit-lemma member-occur)
   (disable-theory sets-and-bags)))



(prove-lemma subbagp-transitive
             (rewrite)
             (implies (and (subbagp x y) (subbagp y z))
                      (subbagp x z))
             ((use (subbagp-necc (a (badguy x z))
                                 (y z)
                                 (x y))
                   (subbagp-necc (a (badguy x z))))
              (disable subbagp-necc)
              (disable-theory sets-and-bags)))


(prove-lemma not-member-make-negs-fact (rewrite)
             (implies
              (not (member x y))
              (equal
               (delete (list 'rneg x) (make-negs y))
               (make-negs y))))

(prove-lemma make-negs-delete (rewrite)
             (equal
              (make-negs (delete a x))
              (delete (list 'rneg a) (make-negs x))))

(prove-lemma make-negs-bagdiff (rewrite)
             (equal
              (make-negs (bagdiff x y))
              (bagdiff (make-negs x) (make-negs y))))
```

```
(prove-lemma append-bagdiff-arg1 (rewrite)
             (implies
              (subbagp a b)
              (equal
               (append (bagdiff b a) c)
               (bagdiff (append b c) a))))

(prove-lemma append-bagdiff-arg2 (rewrite)
             (implies
              (equal (bagint a c) nil)
              (equal
               (append c (bagdiff b a))
               (bagdiff (append c b) a))))


(prove-lemma bagdiff-bagdiff (rewrite)
             (equal
              (bagdiff (bagdiff a b) c)
              (bagdiff a (append b c))))

(prove-lemma member-rneg-make-negs (rewrite)
             (equal
              (member (list 'rneg x) (make-negs y))
              (member x y)))


(prove-lemma subbagp-make-negs (rewrite)
             (equal
              (subbagp (make-negs x) (make-negs y))
              (subbagp x y)))

;(disable rplus-rdifference-arg1)
;(disable rplus-rdifference-arg2)
;(disable rplus-rneg-arg1)
;(disable rplus-rneg-arg2)
;(disable rtimes-rdifference-arg1)
;(disable rtimes-rdifference-arg2)
;(disable rneg-rdifference)
;(disable rdifference-rneg-arg2)

(prove-lemma rdifference-reduce (rewrite)
             (and
              (equal
               (rdifference (reduce x) y)
               (rdifference x y))
              (equal
               (rdifference x (reduce y))
               (rdifference x y)))
             ((enable rdifference)
              (enable-theory r1)
              (disable rplus-rdifference-arg1 rplus-rdifference-arg2
                       rplus-rneg-arg1 rplus-rneg-arg2)))

(prove-lemma requal-simple-rplus-x-simple-rneg-x (rewrite)
             (requal (simple-rplus x (simple-rneg x))
                     (rational 0 1))
             ((enable-theory r1)
              (enable simple-rplus simple-rneg ineg iplus itimes requal
                      fix-rational)))

(lemma requal-rplus-x-simple-rneg-x (rewrite)
       (requal (rplus x (simple-rneg x))
               (rational 0 1))
       ((enable requal-simple-rplus-x-simple-rneg-x rplus
                requal-reduce1)))
```

```
(lemma rplus-x-rneg-x (rewrite)
       (equal (rplus x (rneg x))
              (rational 0 1))
       ((enable rneg rplus-reduce-arg2-rewrite requal-reduce-reduce-equal
                reduce-rplus *1*reduce)
        (use (requal-rplus-x-simple-rneg-x))))

(prove-lemma rdifference-x-x (rewrite)
             (equal
              (rdifference x x)
              (rational 0 1))
             ((enable rdifference)
              (enable-theory r1)
              (disable rplus-rdifference-arg1 rplus-rdifference-arg2
                       rplus-rneg-arg1 rplus-rneg-arg2)))

(prove-lemma rdifference-rplus-hack
             (rewrite)
             (equal (rdifference (rplus a b) a)
                    (reduce b))
             )

(lemma rdifference-rplus-hack2 (rewrite)
       (equal (rdifference (rplus b a) a)
              (reduce b))
       ((enable rdifference-rplus-hack commutativity-of-rplus)))

(prove-lemma rplus-rdifference-hack (rewrite)
             (and
              (equal
               (rplus a (rdifference b a))
               (reduce b))
              (equal
               (rplus (rdifference b a) a)
               (reduce b)))
             ((enable-theory r1)))


(prove-lemma equal-difference-hack1 (rewrite)
             (implies
              (equal a (rdifference b c))
              (equal (rplus c a) (reduce b)))
             ((enable-theory r1)))

(prove-lemma equal-difference-hack2
             (rewrite)
             (implies (equal (reduce a) (rdifference b c))
                      (equal (rplus c a) (reduce b)))
             )

(prove-lemma equal-difference-rewrite (rewrite)
             (implies
              (equal (fix b) (fix d))
             (equal
              (equal (difference a b) (difference c d))
              (or
               (and
                (not (lessp b a))
                (not (lessp b c)))
               (equal a c)))))

(prove-lemma equal-difference-rewrite2 (rewrite)
             (implies
              (equal (fix a) (fix c))
              (equal
               (equal (difference a b) (difference c d))
               (or
                (and
                 (not (lessp b a))
                 (not (lessp d c)))
                (equal (fix b) (fix d))))))

(prove-lemma equal-times-bridge5 (rewrite)
             (implies
              (and
               (equal (times b a) (times c d))
               (equal (times x a) (times c y))
               (not (zerop a)))
              (equal (equal (times b y) (times x d)) t))
              ((enable equal-times-bridge1)))

(prove-lemma equal-plus-difference-rewrite (rewrite)
             (implies
              (equal (fix a) (fix c))
              (and
               (equal
                (equal (plus a b) (difference c d))
                (and
                 (zerop b)
                 (or
                  (zerop a)
                  (zerop d))))
               (equal
                (equal (plus b a) (difference c d))
                (and
                 (zerop b)
                 (or
                  (zerop a)
                  (zerop d))))
               (equal
                (equal (difference c d) (plus a b))
                (and
                 (zerop b)
                 (or
                  (zerop a)
                  (zerop d))))
               (equal
                (equal (difference c d) (plus b a))
                (and
                 (zerop b)
                 (or
                  (zerop a)
                  (zerop d)))))))


(prove-lemma lessp-times-bridge1
             (rewrite)
             (implies (and (lessp (times c v) (times x1 z1))
                           (equal (times w x1) (times c d))
                           (not (zerop c))
                           (not (zerop d)))
                      (lessp (times v w) (times d z1)))
             )
```

```
(prove-lemma equal-difference (rewrite)
             (and
              (equal
               (equal a (difference a b))
               (and
                (numberp a)
                (or
                 (zerop a)
                 (zerop b))))
              (equal
               (equal (difference a b) a)
               (and
                (numberp a)
                (or
                 (zerop a)
                 (zerop b))))))

(prove-lemma requal-simple-rplus-x-x-rewrite (rewrite)
             (equal
              (requal (simple-rplus x y) (simple-rplus x z))
              (requal y z))
             ((enable simple-rplus fix-rational iplus itimes
                      fix-int-on-integers
                      integerp-minus requal equal-times-bridge1
                      equal-times-bridge2 equal-times-bridge3
                      equal-times-bridge4)
              (enable-theory r1)))

(lemma equal-rplus-x-x-rewrite (rewrite)
       (equal
        (equal (rplus x y) (rplus x z))
        (equal (reduce y) (reduce z)))
       ((use (requal-simple-rplus-x-x-rewrite))
        (enable requal-reduce-reduce-equal rplus)))

(prove-lemma equal-rplus-rdifference-hack (rewrite)
             (equal
              (equal (rplus x y) (rdifference (rplus x z) v))
              (equal (reduce y) (rdifference z v)))
             ((enable rdifference associativity-of-rplus reduce-rplus)))


(prove-lemma eval$-rplus-tree-delete (rewrite)
             (equal
              (eval$ t (rplus-tree (delete m x)) a)
              (if (member m x)
                  (rdifference (eval$ t (rplus-tree x) a)
                               (eval$ t m a))
                (eval$ t (rplus-tree x) a)))
             ((enable-theory r1)))


(prove-lemma permutation-does-not-effect-rplus (rewrite)
             (implies
              (permutation x y)
              (equal
               (eval$ t (rplus-tree x) a)
               (eval$ t (rplus-tree y) a)))
             ((enable-theory r1)))

(prove-lemma eval$-rplus-tree-zero (rewrite)
             (equal
              (eval$ t (rplus-tree (append (make-negs x) x)) a)
              (rational 0 1))
             ((enable-theory r1)))

(prove-lemma cancel-zero-fact (rewrite)
             (implies
              (equal (eval$ t (rplus-tree z) a) (rational 0 1))
              (equal
               (eval$ t (rplus-tree (append x z)) a)
               (eval$ t (rplus-tree x) a))))

(prove-lemma member-car-x-x (rewrite)
             (equal
              (member (car x) x)
              (listp x)))

(prove-lemma member-bagdiff-append (rewrite)
             (equal
              (member e (bagdiff (append x z) z))
              (member e x)))

(prove-lemma permutation-transitive (rewrite)
             (implies
              (and
               (permutation x y)
               (permutation y z))
              (permutation x z))
             ((enable permutation-as-subbagp)))

(defn last-cdr (x)
  (if (listp x)
      (last-cdr (cdr x))
    x))

(prove-lemma bagdiff-x-x (rewrite)
             (equal
              (bagdiff x x)
              (last-cdr x)))


(prove-lemma bagdiff-append-arg1 (rewrite)
             (equal
              (bagdiff (append z x) z)
              x))

(prove-lemma bagdiff-cons-z-z (rewrite)
             (equal
              (bagdiff (cons x z) z)
              (cons x (last-cdr z))))


(prove-lemma bagdiff-not-listp (rewrite)
             (implies
              (not (listp x))
              (equal (bagdiff x z) x)))

(prove-lemma bagdiff-car-in (rewrite)
             (implies
              (and
               (listp x)
               (member (car x) z))
              (equal
               (bagdiff x z)
               (bagdiff (cdr x) (delete (car x) z)))))


(prove-lemma last-cdr-delete (rewrite)
             (equal
              (last-cdr (delete e x))
              (last-cdr x)))
```

```
(prove-lemma member-subbagp2 (rewrite)
             (implies
              (and
               (subbagp x y)
               (member e x))
              (member e y))
             ((enable subbagp-wit-lemma member-occur)
              (disable-theory sets-and-bags)))

(prove-lemma member-subbagp-delete (rewrite)
             (implies
              (not (member e x))
              (equal
               (subbagp x (delete e y))
               (subbagp x y))))

(prove-lemma subbagp-bagdiff (rewrite)
             (implies
              (subbagp x y)
              (subbagp (bagdiff x z) (bagdiff y z))))


(prove-lemma permutation-bagdiff (rewrite)
             (implies
              (permutation x y)
              (permutation (bagdiff x z) (bagdiff y z)))
             ((enable permutation-as-subbagp)))


(lemma permutation-append-arg1-arg2-bridge (rewrite)
       (permutation
        (bagdiff (append x z) z)
        (bagdiff (append z x) z))
       ((enable permutation-bagdiff permutation-a-b)))

(prove-lemma subbagp-transitive-bridge-helper (rewrite)
             (implies
              (and
               (subbagp y z)
               (subbagp z y))
              (and
               (equal
                (iff (subbagp y x) (subbagp z x))
                t)
               (equal
                (iff (subbagp x y) (subbagp x z))
                t))))

(lemma subbagp-transitive-bridge (rewrite)
       (implies
        (and
         (subbagp y z)
         (subbagp z y))
        (and
         (equal
          (equal (subbagp y x) (subbagp z x))
          t)
         (equal
          (equal (subbagp x y) (subbagp x z))
          t)))
       ((use (subbagp-transitive-bridge-helper))))

(prove-lemma equal-permutation (rewrite)
             (and
              (implies
               (permutation y z)
               (equal
                (equal (permutation x y) (permutation x z))
                t))
              (implies
               (permutation y z)
               (equal
                (equal (permutation y x) (permutation z x))
                t)))
             ((enable permutation-as-subbagp)))

(lemma permutation-bagdiff-append-helper (rewrite)
       (and
        (equal
         (permutation (bagdiff (append z x) x) y)
         (permutation (bagdiff (append x z) x) y))
        (equal
         (permutation y (bagdiff (append z x) x))
         (permutation y (bagdiff (append x z) x))))
       ((enable equal-permutation permutation-append-arg1-arg2-bridge)))


#|
(prove-lemma bagdiff-append-arg2 (rewrite)
             (equal
              (bagdiff (append x z) z)
              (if (listp x)
                  (append (bagdiff x z) (append (bagint x z) (last-cdr z)))
                (last-cdr z))))

|#

(prove-lemma permutation-bagdiff-append (rewrite)
             (permutation x (bagdiff (append x z) z)))


(prove-lemma cancel-zero-fact-bridge
             (rewrite)
             (implies (and (subbagp z x)
                           (equal (eval$ t (rplus-tree z) a)
                                  (rational 0 1)))
                      (equal (eval$ t (rplus-tree (bagdiff x z)) a)
                             (eval$ t (rplus-tree x) a)))
             )

(prove-lemma rnegs-cancel-list (rewrite)
             (equal
              (eval$ t (rplus-tree (append x (make-negs x))) a)
              (rational 0 1))
             ((enable-theory r1)))

(prove-lemma subbagp-append-simplify1 (rewrite)
             (implies
              (subbagp a x)
              (subbagp a (append x y))))

(prove-lemma subbagp-permutation-equiv (rewrite)
             (implies
              (and
               (permutation x y)
               (subbagp a x))
              (subbagp a y))
              ((enable permutation-as-subbagp)))
```

```
(prove-lemma subbagp-append-simplify2 (rewrite)
             (implies
              (subbagp a x)
              (subbagp a (append y x)))
     ((use (subbagp-permutation-equiv (x (append x y)) (y (append y x))))
              (disable subbagp-permutation-equiv)))

(prove-lemma subbagp-append-bridge (rewrite)
             (implies
              (and
               (subbagp x b)
               (subbagp y a))
              (subbagp (append x y) (append b a))))

(prove-lemma subbagp-append-bridge2 (rewrite)
             (implies
              (and
               (subbagp x b)
               (subbagp y a))
              (subbagp (append x y) (append a b)))
             ((use (subbagp-permutation-equiv (a (append x y)) (x (append b a))
                                              (y (append a b))))))

(prove-lemma cancelling-from-rplus (rewrite)
             (implies
              (and
               (subbagp z x)
               (subbagp z y)
               (equal (bagint (make-negs z) x) nil))
              (equal
               (eval$ t (rplus-tree (append (bagdiff x z)
                        (make-negs (bagdiff y z)))) a)
               (eval$ t (rplus-tree (append x (make-negs y))) a)))
             ((disable rplus-eval$-rplus-tree)))

(prove-lemma eval$-rplus-tree-rplus-fringe (rewrite)
             (implies
              (equal (car x) 'rplus)
              (equal
               (eval$ t (rplus-tree (rplus-fringe x)) a)
               (eval$ t x a)))
             ((induct (rplus-fringe x))
              (enable-theory r1)))


(prove-lemma subbagp-x-x (rewrite)
             (subbagp x x))

(prove-lemma rnegs-not-car-split-by-parity (rewrite)
             (implies
              (and
               (equal (car x) 'rneg)
               (not (equal (caadr x) 'rneg))
               (equal (cddr x) nil))
              (not (member x (car (split-by-parity y))))))

(prove-lemma permutation-append-split-by-parity-bridge (rewrite)
             (permutation (append (car (split-by-parity x))
                                  (make-negs (cdr (split-by-parity x))))
                          x)
             )

(prove-lemma member-cdr-split-by-parity (rewrite)
             (implies
              (member e (cdr (split-by-parity x)))
              (not (equal (car e) 'rneg))))


(defn all-rnegs (x)
  (if (listp x)
      (and
       (equal (caar x) 'rneg)
       (not (equal (caadar x) 'rneg))
       (equal (cddar x) nil)
       (all-rnegs (cdr x)))
    t))

(prove-lemma all-rnegs-make-negs-bagint-fact (rewrite)
             (implies (and (all-rnegs (make-negs (bagint z w)))
                           (member v w))
                      (all-rnegs (make-negs (bagint z (delete v w))))))

(prove-lemma all-rnegs-make-negs-bagint (rewrite)
             (all-rnegs (make-negs (bagint (car (split-by-parity x))
                                           (cdr (split-by-parity y))))))


(prove-lemma bagint-all-rnegs-car-split-by-parity (rewrite)
             (implies
              (all-rnegs x)
             (equal
              (bagint x (car (split-by-parity y)))
              nil)))


(lemma bagint-make-negs-split-by-parity (rewrite)
       (equal (bagint (make-negs (bagint (car (split-by-parity y))
                                         (cdr (split-by-parity y))))
                      (car (split-by-parity y)))
              nil)
       ((enable bagint-all-rnegs-car-split-by-parity
                all-rnegs-make-negs-bagint)))

(prove-lemma correctness-of-cancel-rplus
             ((meta rplus))
             (equal (eval$ t x a)
                    (eval$ t (cancel-rplus x) a))
             )

(lemma commutativity2-of-rtimes (rewrite)
       (equal (rtimes x (rtimes y z))
              (rtimes y (rtimes x z)))
       ((use (associativity-of-rtimes (x x) (y z) (z y)))
        (enable commutativity-of-rtimes)))

(disable CORRECTNESS-OF-CANCEL-RPLUS)
(disable CANCEL-RPLUS)
(disable EVAL$-RPLUS-TREE-RPLUS-FRINGE)
(disable CANCELLING-FROM-RPLUS)
(disable RNEGS-CANCEL-LIST)
(disable CANCEL-ZERO-FACT-BRIDGE)
(disable CANCEL-ZERO-FACT)
(disable EVAL$-RPLUS-TREE-ZERO)
(disable PERMUTATION-DOES-NOT-EFFECT-RPLUS)
(disable EVAL$-RPLUS-TREE-DELETE)
(disable RPLUS-EVAL$-RPLUS-TREE)
(disable REDUCE-EVAL$-RPLUS-TREE)
(disable BAGINT-MAKE-NEGS-SPLIT-BY-PARITY)
(disable BAGINT-ALL-RNEGS-CAR-SPLIT-BY-PARITY)
(disable ALL-RNEGS-MAKE-NEGS-BAGINT)
```

```
(disable ALL-RNEGS-MAKE-NEGS-BAGINT-FACT)
(disable MEMBER-CDR-SPLIT-BY-PARITY)
(disable PERMUTATION-APPEND-SPLIT-BY-PARITY-BRIDGE)
(disable RNEGS-NOT-CAR-SPLIT-BY-PARITY)
(disable SUBBAGP-X-X)
(disable SUBBAGP-APPEND-BRIDGE2)
(disable SUBBAGP-APPEND-BRIDGE)
(disable SUBBAGP-APPEND-SIMPLIFY2)
(disable SUBBAGP-PERMUTATION-EQUIV)
(disable SUBBAGP-APPEND-SIMPLIFY1)
(disable PERMUTATION-BAGDIFF-APPEND)
(disable PERMUTATION-BAGDIFF-APPEND-HELPER)
(disable EQUAL-PERMUTATION)
(disable SUBBAGP-TRANSITIVE-BRIDGE)
(disable SUBBAGP-TRANSITIVE-BRIDGE-HELPER)
(disable PERMUTATION-APPEND-ARG1-ARG2-BRIDGE)
(disable PERMUTATION-BAGDIFF)
(disable SUBBAGP-BAGDIFF)
(disable MEMBER-SUBBAGP-DELETE)
(disable MEMBER-SUBBAGP2)
(disable LAST-CDR-DELETE)
(disable BAGDIFF-CAR-IN)
(disable BAGDIFF-NOT-LISTP)
(disable BAGDIFF-CONS-Z-Z)
(disable BAGDIFF-APPEND-ARG1)
(disable BAGDIFF-X-X)
(disable PERMUTATION-TRANSITIVE)
(disable MEMBER-BAGDIFF-APPEND)
(disable MEMBER-CAR-X-X)
(disable EQUAL-RPLUS-RDIFFERENCE-HACK)
(disable EQUAL-RPLUS-X-X-REWRITE)
(disable REQUAL-SIMPLE-RPLUS-X-X-REWRITE)
(disable EQUAL-DIFFERENCE)
(disable LESSP-TIMES-BRIDGE1)
(disable EQUAL-PLUS-DIFFERENCE-REWRITE)
(disable EQUAL-TIMES-BRIDGE5)
(disable EQUAL-DIFFERENCE-REWRITE2)
(disable EQUAL-DIFFERENCE-REWRITE)
(disable EQUAL-DIFFERENCE-HACK2)
(disable EQUAL-DIFFERENCE-HACK1)
(disable RPLUS-RDIFFERENCE-HACK)
(disable RDIFFERENCE-RPLUS-HACK2)
(disable RDIFFERENCE-RPLUS-HACK)
(disable RDIFFERENCE-X-X)
(disable RPLUS-X-RNEG-X)
(disable REQUAL-RPLUS-X-SIMPLE-RNEG-X)
(disable REQUAL-SIMPLE-RPLUS-X-SIMPLE-RNEG-X)
(disable RDIFFERENCE-REDUCE)
(disable SUBBAGP-MAKE-NEGS)
(disable MEMBER-RNEG-MAKE-NEGS)
(disable BAGDIFF-BAGDIFF)
(disable APPEND-BAGDIFF-ARG2)
(disable APPEND-BAGDIFF-ARG1)
(disable MAKE-NEGS-BAGDIFF)
(disable MAKE-NEGS-DELETE)
(disable NOT-MEMBER-MAKE-NEGS-FACT)
(disable SUBBAGP-TRANSITIVE)
(disable SUBBAGP-NECC)
(disable NOT-SUBBAGP-NOT-PERMUTATION)
(disable PERMUTATION-A-B)
(disable SUBBAGP-PERMUTATION)
(disable SUBBAGP-DELETE-CAR2)
(disable SUBBAGP-DELETE-CAR)
(disable SUBBAGP-DELETE-SAME-MEANS)
(disable SUBBAGP-DELETE-SAME)
(disable SUBBAGP-APPEND)
(disable OCCURRENCES-APPEND)
(disable OCCURRENCES-DELETE2)
```

```
(disable MEMBER-SUBBAGP)
(disable DELETE-APPEND)
(disable MEMBER-APPEND)
(disable EVAL$-REDUCE)
(disable EVAL$-RPLUS)
(disable RPLUS-RZEROP)
(disable RPLUS-RZEROP-BRIDGE2)
(disable RPLUS-RZEROP-BRIDGE)

(deftheory r2
  (
   rtimes-rneg-arg2 rtimes-rneg-arg1 rtimes-rplus-arg2
   rtimes-rplus-arg1  associativity-of-rtimes
    rzerop commutativity-of-rtimes
    rdifference-rdifference-arg2
    rdifference-rdifference-arg1 rneg-rplus
    rplus-reduce-arg2-rewrite rplus-reduce-arg1-rewrite
    reduce-rneg reduce-rmagnitude reduce-rquotient reduce-difference
    reduce-rtimes reduce-rplus
    commutativity2-of-rplus associativity-of-rplus
    equal-requal-rewrite transitivity-of-requal  fix-rational-of-rationalp
    nrational-rplus-arg2  nrational-rplus-arg1  rationalp-means means-rationalp
    rneg-rneg rneg-reduce reduce-0  numberp-numerator-reduce  reduce-nrationalp
    rplus-reduce-arg2   rplus-reduce-arg1   requal-x-x   rplus-requal-arg1
    commutativity-of-rplus   reduce-reduce   requal-reduce2   requal-reduce1
    commutativity-of-requal   rational-generalization   fix-rational-rmagnitude
    fix-rational-rquotient   fix-rational-rdifference   fix-rational-rneg
    fix-rational-rtimes   fix-rational-fix-rational   fix-rational-rplus
    fix-rational-reduce   rationalp-rmagnitude   rationalp-rquotient
    rationalp-rdifference   rationalp-rneg   rationalp-rtimes
    rationalp-fix-rational   rationalp-rplus   rationalp-reduce
    commutativity2-of-rtimes
    rdifference
    correctness-of-cancel-rplus))
```

Appendix C
Floating-point Axiomatization

This appendix lists the forms that introduce the floating-point axioms in a manner guarenteed to be consistent. Some of the events use proof-checker instructions as hints to the prover [12]. These hints have been removed from this listing in the interest of space.

```
(prove-lemma requal-rationalp-non-zero-numerator (rewrite)
             (and
              (implies
               (and
                (requal x y)
                (not (rzerop x)))
               (rationalp y))
              (implies
               (and
                (requal y x)
                (not (rzerop x)))
               (rationalp y)))
             ((enable-theory r1)
              (enable rationalp requal fix-rational itimes fix-int)))

(prove-lemma lessp-times-bridge-bridge
             (rewrite)
             (implies (and (lessp (times a b) (times c d))
                           (lessp (times c x) (times a y)))
                      (lessp (times b x) (times d y)))
             )

(disable lessp-times-bridge-bridge)

(lemma lessp-times-bridge (rewrite)
       (and
        (implies
         (and
          (lessp (times a b) (times c d))
          (lessp (times c x) (times a y)))
         (lessp (times b x) (times d y)))
        (implies
         (and
          (lessp (times b a) (times c d))
          (lessp (times c x) (times a y)))
         (lessp (times b x) (times d y)))
        (implies
         (and
          (lessp (times a b) (times d c))
          (lessp (times c x) (times a y)))
         (lessp (times b x) (times d y)))
        (implies
         (and
          (lessp (times b a) (times d c))
          (lessp (times c x) (times a y)))
         (lessp (times b x) (times d y))))
       ((enable commutativity-of-times lessp-times-bridge-bridge)))
```

```
(prove-lemma lessp-times-bridge-bridge2
             (rewrite)
             (implies (and (lessp (times a b) (times c d))
                           (not (lessp (times a y) (times c x)))
                           (not (equal y 0))
                           (numberp y))
                      (lessp (times b x) (times d y)))
             )


(lemma lessp-times-bridge2 (rewrite)
       (and
        (implies
         (and (lessp (times a b) (times c d))
              (not (lessp (times a y) (times c x)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))
        (implies
         (and (lessp (times a b) (times c d))
              (not (lessp (times y a) (times c x)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))
        (implies
         (and (lessp (times a b) (times c d))
              (not (lessp (times a y) (times x c)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))
        (implies
         (and (lessp (times a b) (times c d))
              (not (lessp (times y a) (times x c)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))

        (implies
         (and (lessp (times b a) (times c d))
              (not (lessp (times a y) (times c x)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))
        (implies
         (and (lessp (times b a) (times c d))
              (not (lessp (times y a) (times c x)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))
        (implies
         (and (lessp (times b a) (times c d))
              (not (lessp (times a y) (times x c)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y)))
        (implies
         (and (lessp (times b a) (times c d))
              (not (lessp (times y a) (times x c)))
              (not (equal y 0))
              (numberp y))
         (lessp (times b x) (times d y))))
       ((enable commutativity-of-times lessp-times-bridge-bridge2)))
```

```
(prove-lemma lessp-times-bridge-bridge3
             (rewrite)
             (implies (and (not (lessp (times a b) (times c d)))
                           (not (lessp (times c x) (times a y)))
                           (numberp a)
                           (not (equal a 0)))
                      (not (lessp (times b x) (times d y))))
             )


(lemma lessp-times-bridge3 (rewrite)
       (and
        (implies
         (and
          (not (lessp (times a b) (times c d)))
          (not (lessp (times c x) (times a y)))
          (numberp a)
          (not (equal a 0)))
         (not (lessp (times b x) (times d y))))
        (implies
         (and
          (not (lessp (times b a) (times c d)))
          (not (lessp (times c x) (times a y)))
          (numberp a)
          (not (equal a 0)))
         (not (lessp (times b x) (times d y))))
        (implies
         (and
          (not (lessp (times a b) (times d c)))
          (not (lessp (times c x) (times a y)))
          (numberp a)
          (not (equal a 0)))
         (not (lessp (times b x) (times d y))))
        (implies
         (and
          (not (lessp (times b a) (times d c)))
          (not (lessp (times c x) (times a y)))
          (numberp a)
          (not (equal a 0)))
         (not (lessp (times b x) (times d y)))))
       ((enable commutativity-of-times lessp-times-bridge-bridge3)))

(prove-lemma rationalp-non-integer (rewrite)
             (implies
              (not (integerp (numerator x)))
              (not (rationalp x)))
             ((enable rationalp)
              (disable integerp)))

(prove-lemma rationalp-zerop (rewrite)
             (implies
              (zerop (denominator x))
              (not (rationalp x)))
             ((enable rationalp)))

(prove-lemma rationalp-not-rational-formp (rewrite)
             (implies
              (not (rational-formp x))
              (not (rationalp x)))
             ((enable rationalp)))
```

```
(prove-lemma rlessp-transitive (rewrite)
             (and
             (implies
              (and (rlessp a b)
                   (rlessp b c))
              (rlessp a c))
             (implies
              (and (rlessp a b)
                   (not (rlessp c b)))
              (rlessp a c))
             (implies
              (and (not (rlessp b a))
                   (rlessp b c))
              (rlessp a c))
             (implies
              (and (not (rlessp b a))
                   (not (rlessp c b)))
              (not (rlessp c a))))
             ((enable rlessp ilessp itimes fix-int fix-rational
                      rationalp-means rational-generalization)))

(prove-lemma rlessp-x-x (rewrite)
             (not (rlessp x x))
             ((enable rlessp ilessp)))

(prove-lemma not-rlessp-if-rlessp (rewrite)
             (implies
              (rlessp x y)
              (not (rlessp y x)))
             ((enable rlessp ilessp)))

(prove-lemma not-rlessp-requal (rewrite)
             (implies
              (requal x y)
              (and
               (not (rlessp x y))
               (not (rlessp y x))))
             ((enable rlessp ilessp requal itimes)))

(prove-lemma fix-int-numerator (rewrite)
             (implies
              (rationalp x)
              (equal (fix-int (numerator x))
                     (numerator x)))
             ((enable fix-int rationalp)
              (enable-theory r1)))

(prove-lemma simple-rplus-0 (rewrite)
             (equal
              (numerator (simple-rplus x (simple-rneg x)))
              0)
             ((enable simple-rplus simple-rneg ineg fix-rational rationalp
                      itimes iplus)
              (enable-theory r1)))

(prove-lemma requal-simple-rdifference-x-x (rewrite)
             (requal (rdifference x x) (rational 0 1))
             ((enable requal rdifference rplus rneg
                      requal-simple-rplus-bridge requal-reduce1
                      fix-int-on-integers fix-rational-simple-rplus
                      rationalp-simple-rplus)
              (enable-theory r1)))

(lemma reduce-rdifference (rewrite)
       (equal
        (reduce (rdifference x y))
        (rdifference x y))
       ((enable reduce-reduce rdifference rplus)))

#
|
(lemma rdifference-x-x (rewrite)
       (equal (rdifference x x) (rational 0 1))
       ((use (requal-simple-rdifference-x-x))
        (enable reduce-rdifference requal-reduce-reduce-equal reduce-0)))
|
#
(prove-lemma requal-fix-rational (rewrite)
             (and
              (requal x (fix-rational x))
              (requal (fix-rational x) x))
             ((enable-theory r1)
              (enable requal)))


(prove-lemma negativep-numerator-reduce (rewrite)
             (equal
              (negativep (numerator (reduce x)))
              (negativep (numerator (fix-rational x))))
             ((enable reduce fix-rational)))

(prove-lemma negativep-numerator-fix-rational (rewrite)
             (equal
              (negativep (numerator (fix-rational x)))
              (and
               (rationalp x)
               (negativep (numerator x))))
             ((enable fix-rational)))

(prove-lemma rneg-fix-rational (rewrite)
             (equal (rneg (fix-rational x))
                    (rneg x))
             ((enable rneg fix-rational-fix-rational simple-rneg)))


(prove-lemma nrationalp-fix-rational (rewrite)
             (implies
              (not (rationalp x))
              (equal (fix-rational x) (rational 0 1)))
             ((enable fix-rational)))

(prove-lemma requal-simple-rmagnitude-reduce (rewrite)
             (requal
              (simple-rmagnitude (reduce x))
              (simple-rmagnitude x))
             ((enable simple-rmagnitude)
              (enable-theory r1)))

(lemma rmagnitude-reduce (rewrite)
       (equal (rmagnitude (reduce x)) (rmagnitude x))
       ((use (requal-simple-rmagnitude-reduce))
        (enable requal-reduce-reduce-equal rmagnitude)))

(prove-lemma numberp-numerator-fix-rational (rewrite)
             (equal
              (numberp (numerator (fix-rational x)))
              (or
               (not (rationalp x))
               (numberp (numerator x))))
             ((enable fix-rational)))
```

```
(prove-lemma requal-rlessp-transitive (rewrite)
             (and
              (implies
               (and
                (requal y x)
                (rlessp x z))
               (rlessp y z))
              (implies
               (and
                (requal y x)
                (not (rlessp x z)))
               (not (rlessp y z)))
               (implies
                (and
                 (requal y x)
                 (rlessp z x))
                (rlessp z y))
               (implies
                (and
                 (requal y x)
                 (not (rlessp z x)))
                (not (rlessp z y)))
               (implies
                (and
                 (requal x y)
                 (rlessp x z))
                (rlessp y z))
               (implies
                (and
                 (requal x y)
                 (not (rlessp x z)))
                (not (rlessp y z)))
               (implies
                (and
                 (requal x y)
                 (rlessp z x))
                (rlessp z y))
               (implies
                (and
                 (requal x y)
                 (not (rlessp z x)))
                (not (rlessp z y)))))

(prove-lemma reduce-fix-rational (rewrite)
             (equal
              (reduce (fix-rational x))
              (reduce x))
       ((enable reduce fix-rational fix-rational-fix-rational)))

(prove-lemma rmagnitude-positive (rewrite)
             (implies
              (numberp (numerator (fix-rational x)))
              (equal (rmagnitude x) (reduce x)))
             ((enable rmagnitude simple-rmagnitude)
              (enable-theory r1)))

(prove-lemma nrationalp-rneg (rewrite)
             (implies
              (not (rationalp x))
              (equal (rneg x) (rational 0 1)))
             ((enable rneg fix-rational reduce simple-rneg)))

(prove-lemma rmagnitude-negative (rewrite)
             (implies
              (negativep (numerator x))
              (equal (rmagnitude x) (rneg x)))
             ((enable rmagnitude simple-rmagnitude)
              (enable-theory r1)))

(prove-lemma rationalp-0 (rewrite)
             (equal
              (rationalp (rational 0 x))
              (lessp 0 x))
             ((enable rationalp)))


(lemma transitivity-of-sign (rewrite)
       (and
        (implies
         (and
          (requal a b)
          (numberp (numerator a))
          (rationalp b))
         (numberp (numerator b)))
        (implies
         (and
          (requal b a)
          (numberp (numerator a))
          (rationalp b))
         (numberp (numerator b))))
        ((enable requal itimes fix-rational fix-int-on-integers rationalp
                 integerp-minus fix-int integerp rational-generalization)
         (enable-theory arithmetic integers rational-defns)))


(prove-lemma lessp-quotient-gcd-bridge (rewrite)
             (implies
              (and
               (not (zerop e))
               (equal (remainder b e) 0)
               (equal (remainder d e) 0))
             (equal
              (lessp (times a (quotient b e))
                     (times c (quotient d e)))
              (lessp (times a b)
                     (times c d)))))



(prove-lemma rlessp-reduce1 (rewrite)
             (equal
              (rlessp (reduce x) y)
              (rlessp x y))
             ((enable ilessp reduce rlessp fix-rational itimes
                      fix-int integerp)
              (enable-theory r1)))

(prove-lemma rlessp-reduce2 (rewrite)
             (equal
              (rlessp x (reduce y))
              (rlessp x y))
             ((enable ilessp reduce rlessp fix-rational itimes
                      fix-int integerp)
              (enable-theory r1)))


#
```

```
|
(prove-lemma numberp-numerator-fpmaximum (rewrite)
             (numberp (numerator (fpmaximum)))
             ((use (fpmaximum-intro))
              (enable rlessp ilessp fix-int fix-rational integerp itimes)))

(lemma negativep-numerator-fpmaximum (rewrite)
       (not (negativep (numerator (fpmaximum))))
       ((use (numberp-numerator-fpmaximum))))

(prove-lemma numberp-numerator-fpminimum (rewrite)
             (numberp (numerator (fpminimum)))
             ((use (fpminimum-intro))))

(lemma negativep-numerator-fpminimum (rewrite)
       (not (negativep (numerator (fpminimum))))
       ((use (numberp-numerator-fpminimum))))

|
#
(prove-lemma rlessp-rmagnitude (rewrite)
             (and
              (equal (rlessp x (rmagnitude y))
                     (if (numberp (numerator y))
                         (rlessp x (fix-rational y))
                       (rlessp x (rneg y))))
              (equal (rlessp (rmagnitude x) y)
                     (if (numberp (numerator (fix-rational x)))
                         (rlessp (fix-rational x) y)
                       (rlessp (rneg x) y))))
             ((enable rmagnitude simple-rmagnitude)))

(lemma rlessp-fix-rational (rewrite)
       (and
        (equal (rlessp (fix-rational x) y)
               (rlessp x y))
        (equal (rlessp x (fix-rational y))
               (rlessp x y)))
       ((enable fix-rational-fix-rational rlessp)))


(prove-lemma rlessp-neg-pos (rewrite)
              (implies
               (and
                (numberp (numerator (fix-rational x)))
                (negativep (numerator (fix-rational y))))
               (and
                (rlessp y x)
                (not (rlessp x y))))
              ((enable rlessp ilessp itimes fix-int integerp)
               (enable-theory r1)))


(prove-lemma numerator-fix-rational-0 (rewrite)
             (equal (numerator (fix-rational (rational 0 x))) 0)
             ((enable fix-rational)))

(prove-lemma rlessp-0 (rewrite)
             (implies
              (equal (numerator (fix-rational x)) 0)
              (and
               (equal (rlessp x y)
                      (and (numberp (numerator (fix-rational y)))
                           (not (equal (numerator (fix-rational y)) 0))))
               (equal (rlessp y x)
                      (negativep (numerator (fix-rational y))))))
             ((enable rlessp ilessp itimes integerp fix-int)
              (enable-theory r1)))

(prove-lemma requal-neg-pos (rewrite)
             (implies
              (and
               (numberp (numerator (fix-rational x)))
               (negativep (numerator (fix-rational y))))
              (and
               (not (requal x y))
               (not (requal y x))))
             ((enable requal itimes fix-int-on-integers integerp-minus)
              (enable-theory r1)))

(prove-lemma nrationalp-rlessp (rewrite)
             (implies
              (not (rationalp x))
              (and
               (equal (rlessp x y) (rlessp (rational 0 1) y))
               (equal (rlessp y x) (rlessp y (rational 0 1)))))
             ((enable rlessp)))



;(prove-lemma rneg-negates-fact (rewrite)
;            (implies
;             (and
;              (numberp (numerator (fix-rational x)))
;              (not (equal (numerator (rneg x)) 0)))
;             (negativep (numerator (rneg x))))
;            ((enable rneg ineg simple-rneg)
;             (enable-theory r1)))


(prove-lemma negativep-numerator (rewrite)
             (equal
              (negativep (numerator x))
              (not (numberp (numerator x))))
             ((enable-theory r1)))

(prove-lemma rneg-negates-fact (rewrite)
             (implies
              (and
               (numberp (numerator (fix-rational x)))
               (not (equal (numerator (rneg x)) 0)))
              (not (numberp (numerator (rneg x)))))
             ((enable-theory r1)
              (enable ineg simple-rneg rneg)))

(prove-lemma rneg-positive-fact (rewrite)
             (implies
              (numberp (numerator (fix-rational x)))
              (not (rlessp x (rneg x))))
             ((use (rneg-negates-fact))
              (enable-theory r1)
              (enable rneg ineg simple-rneg)))
```

```
(prove-lemma nrationalp-requal (rewrite)
             (implies
              (not (rationalp x))
              (and
               (equal (requal x y) (requal (rational 0 1) y))
               (equal (requal y x) (requal y (rational 0 1)))))
             ((enable requal)))

(prove-lemma requal-0 (rewrite)
             (implies
              (equal (numerator (fix-rational x)) 0)
              (and
               (equal (requal x y)
                      (equal (numerator (fix-rational y)) 0))
               (equal (requal y x)
                      (equal (numerator (fix-rational y)) 0))))
             ((enable requal fix-rational izerop)
              (enable-theory r1)))

(prove-lemma requal-rneg1-rneg2-bridge (rewrite)
             (implies
              (and
               (numberp w)
               (numberp z))
              (equal (requal (rational (minus w) d)
                             (rational (minus z) v))
                     (requal (rational w d)
                             (rational z v))))
             ((enable requal itimes fix-int-on-integers integerp-minus
                      fix-rational rationalp)
              (enable-theory r1)))

(prove-lemma requal-rneg1-rneg2 (rewrite)
             (equal
              (requal (rneg x) (rneg y))
              (requal x y))
             ((enable-theory r1)
              (enable rneg simple-rneg ineg)))

(prove-lemma requal-rneg (rewrite)
             (and
              (implies
               (requal x (rneg y))
               (requal (rneg x) y))
              (implies
               (requal (rneg x) y)
               (requal x (rneg y))))
             ((enable-theory r1)
              (enable rneg simple-rneg ineg)))

(lemma rlessp-rneg-requal-rneg (rewrite)
       (and
        (implies
         (requal x (rneg y))
         (and
          (iff
           (rlessp z (rneg x))
           (rlessp z y))
          (iff
           (rlessp (rneg x) z)
           (rlessp y z))))
        (implies
         (requal (rneg y) x)
         (and
          (iff
           (rlessp z (rneg x))
           (rlessp z y))
          (iff
           (rlessp (rneg x) z)
           (rlessp y z)))))
       ((enable-theory r1)
        (enable not-rlessp-requal rlessp-transitive requal-rneg)))

#
|
(prove-lemma constrain-bridge1 (rewrite)
             (implies
              (requal (rneg (fpminimum)) x)
              (not (rlessp (fpmaximum) x)))
             ((use (rneg-positive-fact (x (fpminimum))))
              (disable rneg-positive-fact)
              (enable fpminimum-intro fpmaximum-intro)))

|#

(prove-lemma requal-rneg-constants (rewrite)
             (and
              (implies
               (requal x (rational 1 1))
               (equal (rneg x) (rational -1 1)))
              (implies
               (requal (rational 1 1) x)
               (equal (rneg x) (rational -1 1)))
              (implies
               (requal (rational 0 y) x)
               (equal (rneg x) (rational 0 1)))
              (implies
               (requal x (rational 0 y))
               (equal (rneg x) (rational 0 1)))
              (implies
               (requal x (rational -1 1))
               (equal (rneg x) (rational 1 1)))
              (implies
               (requal (rational -1 1) x)
               (equal (rneg x) (rational 1 1))))
             ((enable rneg requal simple-rneg ineg fix-int fix-rational
                      reduce izerop itimes)
              (enable-theory r1)))
```

```
(prove-lemma rlessp-rneg-constants (rewrite)
             (and
              (equal
               (rlessp (rneg x) (rational 1 1))
               (rlessp (rational -1 1) x))
              (equal
               (rlessp (rational 1 1) (rneg x))
               (rlessp x (rational -1 1)))
              (equal
               (rlessp (rneg x) (rational -1 1))
               (rlessp (rational 1 1) x))
              (equal
               (rlessp (rational -1 1) (rneg x))
               (rlessp x (rational 1 1)))
              (equal
               (rlessp (rneg x) (rational 0 y))
               (rlessp (rational 0 1) x))
              (equal
               (rlessp (rational 0 y) (rneg x))
               (negativep (numerator (fix-rational x)))))
             ((enable rlessp rneg simple-rneg ineg fix-int fix-rational
                      reduce izerop itimes ilessp)
              (enable-theory r1)))

#
|
(prove-lemma not-zero-fpmaximum (rewrite)
             (not (equal (numerator (fpmaximum)) 0))
             ((use (fpmaximum-intro))
              (enable rlessp ilessp)
              (enable-theory r1)))

(prove-lemma negativep-numerator-rneg-fpmaximum (rewrite)
             (negativep (numerator (rneg (fpmaximum))))
             ((use (fpmaximum-intro))
              (enable rlessp itimes rneg ineg simple-rneg rationalp)
              (enable-theory r1)))

(prove-lemma constrain-bridge2 (rewrite)
             (implies
              (numberp (numerator x))
              (not (requal (rneg (fpmaximum)) x)))
             ((use (fpmaximum-intro))
              (enable-theory r1)))

|
#
(prove-lemma requal-rneg-pos-pos (rewrite)
             (implies
              (and
               (numberp (numerator x))
               (numberp (numerator y)))
              (and
               (equal
                (requal (rneg x) y)
                (and
                 (equal (numerator (fix-rational x)) 0)
                 (equal (numerator (fix-rational y)) 0)))
               (equal
                (requal x (rneg y))
                (and
                 (equal (numerator (fix-rational x)) 0)
                 (equal (numerator (fix-rational y)) 0)))))
             ((enable-theory r1)
              (enable rneg ineg simple-rneg)))

(prove-lemma constrain-bridge3 (rewrite)
             (and
              (equal
               (requal (rneg x) (rational -1 1))
               (requal x (rational 1 1)))
              (equal
               (requal (rational -1 1) (rneg x))
               (requal x (rational 1 1)))
              (equal
               (requal (rneg x) (rational 0 y))
               (requal x (rational 0 1)))
              (equal
               (requal (rational 0 y) (rneg x))
               (requal x (rational 0 1)))
              (equal
               (requal (rneg x) (rational 1 1))
               (requal x (rational -1 1)))
              (equal
               (requal (rational 1 1) (rneg x))
               (requal x (rational -1 1))))
             ((enable requal rneg simple-rneg fix-int fix-rational itimes ineg)
              (enable-theory r1)))

#
|
(prove-lemma rationalp-fpmaximum-minimum (rewrite)
             (and
              (rationalp (fpmaximum))
              (rationalp (fpminimum)))
             ((use (fpmaximum-intro) (fpminimum-intro))))

(prove-lemma constrain-bridge4 (rewrite)
             (and
              (rlessp (fpminimum) (rational 1 1))
              (rlessp (fpminimum) (fpmaximum))
              (not (rlessp (fpminimum) (rational 0 y)))
              (rlessp (rational 0 y) (fpmaximum))
              (rlessp (rational 1 1) (fpmaximum)))
             ((use (fpminimum-intro) (fpmaximum-intro))
              (enable-theory r1)))
|#


(prove-lemma constrain-bridge5 (rewrite)
             (implies
              (and
               (not (rlessp x y))
               (numberp (numerator y))
               (numberp (numerator z))
               (not (equal (numerator (fix-rational z)) 0)))
              (rlessp (rneg x) z))
             ((enable rlessp ilessp itimes rneg ineg simple-rneg
                      fix-int-on-integers integerp-minus)
              (enable-theory r1)))

(prove-lemma rlessp-zero-means-negativep (rewrite)
             (implies
              (rlessp x (rational 0 y))
              (and
               (rationalp x)
               (negativep (numerator x)))))
```

```
(prove-lemma rtimes-1 (rewrite)
             (implies
              (not (zerop x))
              (and
               (equal (rtimes (rational x x) y)
                      (reduce y))
               (equal (rtimes y (rational x x))
                      (reduce y))))
              ((enable rtimes simple-rtimes itimes fix-rational reduce fix-int)
               (enable-theory r1)))

(prove-lemma rtimes-minus-1 (rewrite)
             (implies
              (not (zerop x))
              (and
               (equal (rtimes (rational (minus x) x) y)
                      (rneg y))
               (equal (rtimes y (rational (minus x) x))
                      (rneg y))))
             ((enable rtimes simple-rtimes itimes fix-rational fix-int rneg
                      simple-rneg ineg reduce)
              (enable-theory r1)))

(prove-lemma numberp-numerator-rneg (rewrite)
             (equal
              (numberp (numerator (rneg x)))
              (or (negativep (numerator x))
                  (rzerop x)))
             ((enable rneg simple-rneg ineg)
              (enable-theory r1)))


(prove-lemma negativep-numerator-rneg (rewrite)
             (equal
              (negativep (numerator (rneg x)))
              (rlessp (rational 0 1) x))
             ((enable rneg simple-rneg ineg rlessp ilessp itimes)
              (enable-theory r1)))
#
|
(prove-lemma rplus-rzerop-bridge (rewrite)
             (implies
              (and
               (not (zerop v))
               (numberp z))
              (equal (reduce (rational (times v z) (times v w)))
                     (reduce (rational z w))))
             ((enable reduce rationalp)
              (enable-theory r1)))

(prove-lemma rplus-rzerop-bridge2 (rewrite)
             (implies
              (not (zerop v))
              (equal (reduce (rational (minus (times d v))
                                       (times v w)))
                     (reduce (rational (minus d) w))))
             ((enable reduce rationalp)
              (enable-theory r1)))
```

```
(prove-lemma rplus-rzerop (rewrite)
             (implies
              (rzerop x)
              (and
               (equal (rplus x y) (reduce y))
               (equal (rplus y x) (reduce y))))
             ((enable rplus simple-rplus fix-int-on-integers iplus itimes
                      fix-rational)
              (enable-theory r1)))
|
#
(prove-lemma numerator-fix-rational-zero-rzerop (rewrite)
             (implies
              (equal (numerator (fix-rational x)) 0)
              (rzerop x))
             ((enable fix-rational)
              (enable-theory r1)))

(prove-lemma numerator-reduce-0 (rewrite)
             (equal
              (equal (numerator (reduce x)) 0)
              (or
               (not (rationalp x))
               (equal (numerator x) 0)))
             ((enable reduce)
              (enable-theory r1)))

(prove-lemma constrain-bridge8 (rewrite)
             (and
              (implies
               (and
                (numberp (numerator (fix-rational x)))
                (numberp (numerator (fix-rational y))))
                (equal
                 (equal (numerator (rplus x y)) 0)
                 (and
                  (equal (numerator (fix-rational x)) 0)
                  (equal (numerator (fix-rational y)) 0))))
              (implies
               (and
                (negativep (numerator (fix-rational x)))
                (numberp (numerator (fix-rational y))))
               (and
                (equal
                 (equal (numerator (rplus x y)) 0)
                 (requal (rneg x) y))
                (equal
                 (equal (numerator (rplus y x)) 0)
                 (requal (rneg x) y))))
              (implies
               (and
                (negativep (numerator (fix-rational x)))
                (negativep (numerator (fix-rational y))))
               (not (equal (numerator (rplus x y)) 0))))
             ((enable rplus simple-rplus fix-int izerop itimes ilessp iplus
                     rneg ineg simple-rneg requal)
              (enable-theory r1)))
```

```
(prove-lemma constrain-bridge9-hack
             (rewrite)
             (implies (and (not (zerop z1))
                           (not (zerop a))
                           (equal (times w x1) (times c d)))
                      (equal (equal (times c d z1)
                                    (plus a (times w x1 z1)))
                             f))
             )

(prove-lemma constant-bridge9-hack2
             (rewrite)
             (implies (and (not (zerop z1))
                           (not (zerop a))
                           (not (zerop c))
                           (not (zerop d))
                           (equal (times w x1) (times c d)))
                      (equal (equal (times c d z1)
                                    (difference (times w x1 z1) a))
                             f))
             )

(prove-lemma constrain-bridge9 (rewrite)
             (implies
              (requal x z)
              (equal
               (requal (rplus x y) z)
               (rzerop y)))
             ((enable rplus requal iplus itimes requal simple-rplus izerop
                      fix-int-on-integers integerp-minus)
              (enable-theory r1)))


(lemma constrain-bridge9-variants (rewrite)
       (and
        (implies
         (requal x z)
         (equal
          (requal (rplus y x) z)
          (rzerop y)))
        (implies
         (requal x z)
         (equal
          (requal z (rplus x y))
          (rzerop y)))
        (implies
         (requal x z)
         (equal
          (requal z (rplus y x))
          (rzerop y))))
       ((enable commutativity-of-rplus commutativity-of-requal)
        (use (constrain-bridge9))))

(prove-lemma rplus-positive (rewrite)
             (and
              (implies
               (and
                (numberp (numerator (fix-rational x)))
                (numberp (numerator (fix-rational y))))
               (numberp (numerator (rplus x y))))
              (implies
               (and
                (negativep (numerator (fix-rational x)))
                (numberp (numerator (fix-rational y))))
               (and
                (equal (numberp (numerator (rplus x y)))
                       (not (rlessp y (rneg x))))
                (equal (numberp (numerator (rplus y x)))
                       (not (rlessp y (rneg x))))))
              (implies
               (and
                (negativep (numerator (fix-rational x)))
                (negativep (numerator (fix-rational y))))
               (not (numberp (numerator (rplus x y))))))
             ((enable rplus simple-rplus fix-int izerop itimes ilessp iplus
                      rneg ineg simple-rneg requal rlessp)
              (enable-theory r1)))

;
 just so it’s most recent, redo requal-x-x proof
(lemma requal-x-x-copy (rewrite)
       (requal x x)
       ((enable requal-x-x)))


(prove-lemma integerp-numerator (rewrite)
             (implies
              (rationalp x)
              (integerp (numerator x)))
             ((enable integerp)
              (enable-theory r1)))

;
; silliest lemmas ever
(prove-lemma constrain-super-hack
             (rewrite)
             (implies (and (numberp v)
                           (not (equal v 0))
                           (requal (rational -1 1) x)
                           (numberp z)
                           (not (equal z 0))
                           (rlessp (rational z v)
                                   (rational 1 1)))
                      (not (equal (numerator (rplus x (rational z v)))
                                  0)))
             )

(prove-lemma constrain-super-hack2
             (rewrite)
             (implies (and (numberp v)
                           (not (equal v 0))
                           (requal (rational -1 1) x)
                           (numberp z)
                           (not (equal z 0))
                           (rlessp (rational z v)
                                   (rational 1 1)))
                      (not (requal (rational 1 1)
                                   (rplus x (rational z v)))))
             )
```

```
(constrain fpp-round-intro (rewrite)
           (and
            (fpp (rational 0 1))
            (fpp (rational 1 1))
            (implies (rationalp x) (equal (fpp (reduce x)) (fpp x)))
            (implies (fpp x) (rationalp x)) ; added 3-26-90
            (fpp (fpmaximum))
            (fpp (fpminimum))
            (implies (fpp x)
                     (not (rlessp (fpmaximum) (rmagnitude x))))
            (implies (and (fpp x) (not (equal (numerator (fix-rational x)) 0)))
                     (not (rlessp (rmagnitude x) (fpminimum))))
            (fpp (round x))
            (implies (rationalp x) (equal (fpp (rneg x)) (fpp x)))
            (implies
             (and (fpp y)
                  (not (rlessp x y)))
             (not (rlessp (round x) y)))
            (implies
             (and (fpp y)
                  (not (rlessp y x)))
             (not (rlessp y (round x))))
            (implies
             (and (not (rlessp x (fpminimum)))
                  (not (rlessp (fpmaximum) x)))
             (and
              (not (rlessp (round x) (rtimes (round-min) x)))
              (not (rlessp (rtimes (round-max) x) (round x)))))
           (equal (round (rneg x)) (rneg (round x)))
           (implies
            (and
             (fpp x)
             (numberp (numerator (fix-rational delta)))
             (not (equal (numerator (fix-rational delta)) 0))
             (rlessp delta (fpminspace)))
            (not (fpp (rplus x delta))))
           (rlessp (rational 0 1) (fpminspace))
           (not (rlessp (rational 1 1) (round-min)))
           (rlessp (rational 99 100) (round-min))
           (not (rlessp (round-max) (rational 1 1)))
           (rlessp (round-max) (rational 101 100))
           (numberp (numerator (fpminimum)))
           (not (rlessp (rational 1 1) (fpminimum)))
           (not (rlessp (fpmaximum) (rational 1 1))))

           ((fpp (lambda (x) (and (rationalp x)
                                  (or
                                   (requal x (rational 0 1))
                                   (requal x (rational 1 1))
                                   (requal x (rational -1 1))))))
            (fpminspace (lambda nil (rational 1 1)))
            (fpminimum (lambda nil (rational 1 1)))
            (fpmaximum (lambda nil (rational 1 1)))
            (round-max (lambda nil (rational 1 1)))
            (round-min (lambda nil (rational 1 1)))
            (round (lambda (x) (if (rlessp (rmagnitude x) (rational 1 1))
                                   (rational 0 1)
                                 (if (numberp (numerator x))
                                     (rational 1 1)
                                   (rational -1 1))))))
           ((enable-theory r1) (enable rquotient)))

;Floating-point axioms

(lemma fpp-0 (rewrite)
       (fpp (rational 0 1))
       ((enable fpp-round-intro)))

(lemma fpp-1 (rewrite)
       (fpp (rational 1 1))
       ((enable fpp-round-intro)))

(lemma fpp-reduce (rewrite)
       (implies
        (rationalp x)
        (equal (fpp (reduce x)) (fpp x)))
       ((enable fpp-round-intro)))

(lemma fpp-maximum (rewrite)
       (fpp (fpmaximum))
       ((enable fpp-round-intro)))

(lemma fpp-bounded-fpmaximum (rewrite)
       (implies
        (fpp x)
        (not (rlessp (fpmaximum) (rmagnitude x))))
       ((enable fpp-round-intro)))

(lemma fpp-bounded-fpminimum (rewrite)
       (implies
        (and
         (fpp x)
         (not (equal (numerator (fix-rational x)) 0)))
        (not (rlessp (rmagnitude x) (fpminimum))))
       ((enable fpp-round-intro)))

(lemma fpp-round (rewrite)
       (fpp (round x))
       ((enable fpp-round-intro)))

(lemma fpp-rneg (rewrite)
       (implies (rationalp x) (equal (fpp (rneg x)) (fpp x)))
       ((enable fpp-round-intro)))

(lemma not-round-down-past (rewrite)
       (implies
        (and (fpp y)
             (not (rlessp x y)))
        (not (rlessp (round x) y)))
       ((enable fpp-round-intro)))

(lemma not-round-up-past (rewrite)
       (implies
        (and (fpp y)
             (not (rlessp y x)))
        (not (rlessp y (round x))))
       ((enable fpp-round-intro)))

(lemma fp-round-min-bound (rewrite)
       (implies
        (and
         (not (rlessp x (fpminimum)))
         (not (rlessp (fpmaximum) x)))
        (not (rlessp (round x) (rtimes (round-min) x))))
       ((enable fpp-round-intro)))

(lemma fp-round-max-bound (rewrite)
       (implies
        (and
         (not (rlessp x (fpminimum)))
         (not (rlessp (fpmaximum) x)))
        (not (rlessp (rtimes (round-max) x) (round x))))
       ((enable fpp-round-intro)))
```

```
(lemma rlessp-0-fpminspace (rewrite)
       (rlessp (rational 0 1) (fpminspace))
       ((enable fpp-round-intro)))


(lemma round-rneg (rewrite)
       (equal (round (rneg x)) (rneg (round x)))
       ((enable fpp-round-intro)))

(lemma fpp-minspace (rewrite)
       (implies
        (and
         (fpp x)
         (numberp (numerator (fix-rational delta)))
         (not (equal (numerator (fix-rational delta)) 0))
         (rlessp delta (fpminspace)))
        (not (fpp (rplus x delta))))
       ((enable fpp-round-intro)))

(disable fpp-round-intro)
```

Appendix D
fp-mid and a Correctness Proof

This appendix lists the forms that introduce **fp-mid**, a program that finds the zero of a function, and a proof of its correctness. Some of the events use proof-checker instructions as hints to the prover [12]. These hints have been removed from this listing in the interest of space.

```
#
|;; definitions placed in rational.events - for replay remove
(defn simple-rinverse (r)
  (if (rzerop r)
      (rational 0 1)
  (if (negativep (numerator r))
     (rational (ineg (denominator r))
               (ineg (numerator r)))
    (rational (denominator r) (numerator r)))))

(defn rinverse (r)
  (reduce (simple-rinverse r)))

(definition rquotient2 (x y)
  (rtimes x (rinverse y)))
|#

;; Some arithmetic facts we're missing

(disable rinverse)


(lemma rtimes-reduce (rewrite)
       (and
        (equal (rtimes (reduce x) y)
               (rtimes x y))
        (equal (rtimes x (reduce y))
               (rtimes x y)))
       ((use (requal-simple-rtimes-reduce-arg1)
             (requal-simple-rtimes-reduce-arg2))
        (enable rtimes requal-reduce-reduce-equal)))


(lemma equal-reduce-reduce (rewrite)
       (equal
        (equal (reduce x) (reduce y))
        (requal x y))
       ((use (requal-reduce-reduce-equal (a x) (b y)))))

(prove-lemma times-quotient-gcd-bridge
             (rewrite)
             (equal (equal (times a (quotient b (gcd d b)))
                           (times c (quotient d (gcd d b))))
                    (or (equal (times a b) (times c d))
                        (and (zerop b) (zerop d))))
             )
```

```
(lemma times-quotient-gcd-bridge2 (rewrite)
       (equal
        (equal (times a (quotient b (gcd b d)))
               (times c (quotient d (gcd b d))))
        (or
         (equal (times a b)
                (times c d))
         (and (zerop b) (zerop d))))
       ((enable commutativity-of-gcd times-quotient-gcd-bridge)))

(prove-lemma requal-simple-rinverse-reduce-bridge-helper (rewrite)
             (requal (simple-rinverse (reduce x)) (simple-rinverse x))
             ((enable-theory r2)
              (enable requal itimes fix-int reduce simple-rinverse ineg
                      integerp-minus)))

(prove-lemma requal-simple-rinverse-reduce-bridge (rewrite)
             (and
              (equal
               (requal (simple-rinverse (reduce x)) y)
               (requal (simple-rinverse x) y))
              (equal
               (requal y (simple-rinverse (reduce x)))
               (requal y (simple-rinverse x))))
             ((use (requal-simple-rinverse-reduce-bridge-helper))
              (disable simple-rinverse requal-simple-rinverse-reduce-bridge-helper)
              (enable equal-requal-rewrite commutativity-of-requal)))



(prove-lemma requal-simple-rneg-reduce-bridge-helper (rewrite)
             (requal (simple-rneg (reduce x)) (simple-rneg x))
             ((enable-theory r2)
              (enable requal itimes fix-int reduce simple-rneg ineg
                      integerp-minus)))

(prove-lemma requal-simple-rneg-reduce-bridge (rewrite)
             (and
              (equal
               (requal (simple-rneg (reduce x)) y)
               (requal (simple-rneg x) y))
              (equal
               (requal y (simple-rneg (reduce x)))
               (requal y (simple-rneg x))))
             ((use (requal-simple-rneg-reduce-bridge-helper))
              (disable simple-rneg requal-simple-rneg-reduce-bridge-helper)
              (enable equal-requal-rewrite commutativity-of-requal)))


(prove-lemma requal-simple-rinverse-simple-rneg (rewrite)
             (requal
              (simple-rinverse (simple-rneg x))
              (simple-rneg (simple-rinverse x)))
             ((enable simple-rinverse simple-rneg requal itimes fix-int-on-integers
                      integerp-minus ineg)
              (enable-theory r2)))

(prove-lemma rinverse-rneg (rewrite)
             (equal
              (rinverse (rneg x))
              (rneg (rinverse x)))
             ((enable-theory r2)
              (enable rinverse rneg)
              (disable simple-rinverse simple-rneg)))
```

```
(prove-lemma rationalp-rinverse (rewrite)
             (rationalp (rinverse x))
             ((enable rinverse rationalp-reduce)))

(prove-lemma rinverse-reduce (rewrite)
             (equal
              (rinverse (reduce x))
              (rinverse x))
             ((enable-theory r2)
              (enable rinverse simple-rinverse reduce ineg fix-rational rationalp
                      gcd-quotient-quotient)))

(prove-lemma requal-rtimes-rinverse-arg2 (rewrite)
             (requal
              (simple-rtimes x (simple-rinverse x))
              (if (rzerop x)
                  (rational 0 1)
                (rational 1 1)))
             ((enable-theory r2)
              (enable requal simple-rinverse simple-rtimes itimes
                      fix-int-on-integers integerp-minus ineg)))


(lemma requal-rtimes-rinverse-arg2-bridge (rewrite)
       (requal
        (reduce (rtimes x (rinverse x)))
        (reduce (if (rzerop x)
                    (rational 0 1)
                  (rational 1 1))))
       ((enable rtimes requal-reduce1 requal-reduce2 requal-rtimes-rinverse-arg2
                 rinverse requal-simple-rtimes-bridge)))


(lemma rtimes-rinverse-arg2 (rewrite)
       (equal
        (rtimes x (rinverse x))
        (if (rzerop x)
            (rational 0 1)
          (rational 1 1)))
       ((use (requal-rtimes-rinverse-arg2-bridge))
        (enable requal-reduce-reduce-equal reduce-reduce reduce-rtimes *1*reduce)))

(disable requal-rtimes-rinverse-arg2)
(disable requal-rtimes-rinverse-arg2-bridge)

(prove-lemma negativep-iplus (rewrite)
             (equal
              (negativep (iplus x y))
              (or
               (and
                (negativep x)
                (lessp y (negative-guts x)))
               (and
                (negativep y)
                (lessp x (negative-guts y)))))
             ((enable iplus)))
```

```
(prove-lemma itimes-minus-1 (rewrite)
             (implies
              (lessp 0 x)
              (and
               (equal
                (itimes -1 x)
                (minus x))
               (equal
                (itimes x -1)
                (minus x))))
             ((enable itimes fix-int)))

(prove-lemma iplus-x-minus-z-bridge (rewrite)
             (implies
              (and
               (numberp x)
               (numberp z)
               (not (lessp x z)))
              (equal (iplus x (minus z))
                     (difference x z)))
             ((enable iplus)))

(prove-lemma numberp-numerator-rtimes (rewrite)
             (equal
              (numberp (numerator (rtimes x y)))
              (or
               (rzerop x)
               (rzerop y)
               (equal (numberp (numerator x)) (numberp (numerator y)))))
             ((enable-theory r2)
              (enable rtimes simple-rtimes itimes fix-int-on-integers integerp-minus)))

(prove-lemma numberp-numerator-rinverse (rewrite)
             (equal
              (numberp (numerator (rinverse x)))
              (numberp (numerator (fix-rational x))))
             ((enable-theory r2)
              (enable rinverse simple-rinverse ineg)))

(prove-lemma rzerop-rlessp-rewrite (rewrite)
             (implies
              (rzerop x)
              (and
               (equal
                (rlessp x y)
                (ilessp 0 (numerator (fix-rational y))))
               (equal
                (rlessp y x)
                (negativep (numerator (fix-rational y))))))
             ((enable-theory r2)
              (enable rlessp ilessp itimes fix-rational fix-int integerp-minus)))

(prove-lemma rzerop-rtimes (rewrite)
             (implies
              (rzerop x)
              (and
               (equal (rtimes x y) (rational 0 1))
               (equal (rtimes y x) (rational 0 1))))
             ((enable-theory r2)
              (enable rtimes simple-rtimes)))

(prove-lemma quotient-numerator-rplus-denominator-rplus (rewrite)
             (equal (quotient (numerator (rplus x (rational -1 1)))
                              (denominator (rplus x (rational -1 1))))
                    (sub1 (quotient (numerator (fix-rational x))
                                    (denominator (fix-rational x)))))
             ((enable-theory r2)
              (enable rplus simple-rplus reduce iplus itimes fix-int integerp-minus)))

(lemma quotient-numerator-rplus-denominator-rplus2 (rewrite)
             (equal (quotient (numerator (rplus (rational -1 1) x))
                              (denominator (rplus (rational -1 1) x)))
                    (sub1 (quotient (numerator (fix-rational x))
                                    (denominator (fix-rational x)))))
             ((use (quotient-numerator-rplus-denominator-rplus))
              (enable commutativity-of-rplus)))

(prove-lemma rlessp-neg-pos2 (rewrite)
             (implies
              (and
               (numberp (numerator x))
               (negativep (numerator y))
               (rationalp y))
              (and
               (rlessp y x)
               (not (rlessp x y))))
             ((enable rlessp ilessp itimes fix-int integerp)
              (enable-theory r1)))

;; some new rational functions

(defn rabs (x)
  (if (negativep (numerator x))
      (rneg x)
    (reduce x)))

(defn rmin (x y)
  (if (rlessp x y)
      x
    y))

(defn rmax (x y)
  (if (rlessp x y)
      y
    x))

;;;; The zero-finding problem

(constrain func-intro (rewrite)
           (fpp (func x))
           ((func (lambda (x) (rational 0 1)))))

(defn floor (r)
  (if (and
       (rationalp r)
       (numberp (numerator r)))
      (quotient (numerator r) (denominator r))
    0))
```

```
(defn find-func-zero-measure (a b minspace)
  (floor (rquotient (rdifference b a) minspace)))


(prove-lemma floor-rplus-minus-1 (rewrite)
             (equal
              (floor (rplus x (rational -1 1)))
              (sub1 (floor x)))
             ((enable-theory r2)
              (enable rlessp ilessp fix-int integerp-minus)))

(prove-lemma floor-0 (rewrite)
             (equal
              (equal (floor x) 0)
              (rlessp x (rational 1 1)))
             ((enable-theory r2)
              (enable rlessp ilessp floor fix-int)))


(prove-lemma rlessp-simple-rtimes-reduce-arg1-bridge (rewrite)
              (equal
               (rlessp (rtimes (reduce x) y) z)
               (rlessp (rtimes x y) z))
              ((enable-theory r2)))

(lemma rlessp-simple-rtimes-reduce-arg1 (rewrite)
       (equal
        (rlessp (simple-rtimes (reduce x) y) z)
        (rlessp (simple-rtimes x y) z))
       ((enable-theory r2)
        (use (rlessp-simple-rtimes-reduce-arg1-bridge))
        (enable rtimes rlessp-reduce1)))

(lemma rlessp-simple-rtimes-reduce-arg1-2 (rewrite)
       (equal
        (rlessp (simple-rtimes y (reduce x)) z)
        (rlessp (simple-rtimes y x) z))
       ((use (rlessp-simple-rtimes-reduce-arg1))
        (enable commutativity-of-simple-rtimes)))


(prove-lemma rlessp-simple-rtimes-reduce-arg2-bridge (rewrite)
              (equal
               (rlessp z (rtimes (reduce x) y))
               (rlessp z (rtimes x y)))
              ((enable-theory r2)))

(prove-lemma rlessp-simple-rtimes-reduce-arg2 (rewrite)
              (equal
               (rlessp z (simple-rtimes (reduce x) y))
               (rlessp z (simple-rtimes x y)))
              ((enable-theory r2)
               (use (rlessp-simple-rtimes-reduce-arg2-bridge))
               (enable rtimes rlessp-reduce2)))

(prove-lemma lessp-sub1 (rewrite)
             (equal
              (lessp (sub1 x) x)
              (lessp 0 x)))

(lemma rlessp-simple-rtimes-reduce-arg2-2 (rewrite)
       (equal
        (rlessp z (simple-rtimes y (reduce x)))
        (rlessp z (simple-rtimes y x)))
       ((use (rlessp-simple-rtimes-reduce-arg2))
        (enable commutativity-of-simple-rtimes)))

(prove-lemma rlessp-rtimes-rational-1-1 (rewrite)
             (equal
              (rlessp (rtimes x (rinverse y)) (rational 1 1))
              (or
               (rlessp (rmagnitude x) (rmagnitude y))
               (not (equal (numberp (numerator x)) (numberp (numerator y))))
               (rzerop x)
               (rzerop y)))
             ((enable-theory r2)
              (enable rlessp ilessp rquotient  rtimes simple-rtimes
                      rinverse simple-rinverse itimes fix-int fix-rational rneg ineg
                      simple-rneg)))

(prove-lemma denominator-0 (rewrite)
             (implies
              (rationalp x)
              (lessp 0 (denominator x))))


(prove-lemma numberp-numerator-simple-rplus-x-simple-rneg-y (rewrite)
             (implies
              (not (rlessp x y))
              (numberp (numerator (simple-rplus x (simple-rneg y)))))
             ((enable-theory r2 arithmetic rational-defns)
              (enable rlessp ilessp itimes rplus simple-rplus rneg simple-rneg ineg
                      fix-int iplus *1*fix-rational numberp-numerator-fix-rational
                      integerp-minus implies numberp-numerator-reduce
                      negativep-numerator negativep-numerator-reduce
                      negativep-numerator-fix-rational denominator-0
                      negative-guts-minus integerp zerop and)
              (disable-theory t)))

(prove-lemma numberp-numerator-rplus-reduce (rewrite)
             (equal (numberp (numerator (simple-rplus x (reduce y))))
                    (numberp (numerator (simple-rplus x y))))
             ((enable-theory r2)
              (enable reduce simple-rplus itimes iplus fix-int integerp)))

(lemma numberp-numerator-rplus-x-rneg-y (rewrite)
       (implies
        (not (rlessp x y))
        (numberp (numerator (rplus x (rneg y)))))
       ((enable numberp-numerator-rplus-reduce rplus numberp-numerator-reduce
                numberp-numerator-simple-rplus-x-simple-rneg-y
                fix-rational-simple-rplus rneg)))

(prove-lemma times-zero-2 (rewrite)
             (implies
              (zerop x)
              (equal (times x y) 0)))

(prove-lemma difference-0 (rewrite)
             (equal (difference x 0) (fix x)))

(prove-lemma negativep-fix-int-minus (rewrite)
             (equal
              (negativep (fix-int (minus x)))
              (lessp 0 x))
             ((enable fix-int)))

(prove-lemma fix-int-minus-0 (rewrite)
             (implies
              (zerop x)
              (equal
               (fix-int (minus x))
               0)))
```

```
(prove-lemma lessp-difference-rewrite (rewrite)
             (equal
              (lessp (difference x y) x)
              (and
               (lessp 0 y)
               (lessp 0 x))))

(prove-lemma lessp-difference-difference-arg1-rewrite (rewrite)
             (equal
              (lessp (difference x y)
                     (difference x z))
              (and
               (lessp z x)
               (lessp z y))))

(prove-lemma lessp-difference-difference-arg2-rewrite (rewrite)
              (equal
               (lessp (difference x y)
                      (difference z y))
               (and
                (lessp x z)
                (lessp y z))))

(prove-lemma lessp-difference-plus-rewrite (rewrite)
             (and
              (equal
               (lessp (difference x y)
                      (plus x z))
               (or
                (lessp 0 z)
                (and
                 (not (zerop x))
                 (not (zerop y)))))
              (equal
               (lessp (difference x y)
                      (plus z x))
               (or
                (lessp 0 z)
                (and
                 (not (zerop x))
                 (not (zerop y)))))))

(prove-lemma equal-lessp-bridge
             (rewrite)
             (implies (and (equal (times z zw) (times d v))
                           (lessp (times z1 zw) (times d zx1))
                           (numberp z)
                           (not (equal z 0)))
                      (lessp (times v z1) (times z zx1)))
             )

(prove-lemma rlessp-rplus-rplus (rewrite)
             (equal (rlessp (rplus c x) (rplus c y))
                    (rlessp x y))
             ((enable-theory r2 arithmetic rational-defns)
              (disable-theory t)
              (enable rlessp rplus simple-rplus rlessp-reduce1 rlessp-reduce2
                      ilessp fix-rational itimes lessp zerop times-zero-2
                      iplus fix-int-on-integers integerp integerp-minus
                      difference-0 fix implies and or equal-lessp-bridge
                      lessp-difference-plus-rewrite
                      negative-guts-minus negativep-fix-int-minus
                      lessp-difference-difference-arg1-rewrite
                      lessp-difference-difference-arg2-rewrite
                      fix-int-minus-0 lessp-difference-rewrite
                      lessp-times-bridge correctness-of-cancel-rplus
                      lessp-times-bridge2 lessp-times-bridge3)))

(prove-lemma numerator-reduce-bridge (rewrite)
             (implies
              (equal (numerator (reduce x)) 0)
              (equal (reduce x) (rational 0 1)))
             ((enable-theory r2)))

(prove-lemma numerator-rplus-bridge (rewrite)
             (implies
              (equal (numerator (rplus x y)) 0)
              (equal (rplus x y) (rational 0 1)))
             ((enable rplus numerator-reduce-bridge)))

(lemma negativep-means-not-lessp (rewrite)
       (implies
        (negativep x)
        (equal (lessp 0 x) f)))

(prove-lemma negative-guts-numerator-reduce-x-0 (rewrite)
             (implies
              (and
               (negativep (numerator x))
               (rationalp x))
             (lessp 0 (negative-guts (numerator (reduce x)))))
             ((enable-theory r2)
              (enable reduce)))

(prove-lemma lessp-times-0-rewrite (rewrite)
             (equal
              (lessp 0 (times a b))
              (and
               (lessp 0 a)
               (lessp 0 b))))

(prove-lemma lessp-0-from-not-zerop (rewrite)
             (implies
              (and
               (numberp x)
               (not (equal x 0)))
              (equal (lessp 0 x) t)))

(prove-lemma not-lessp-0-means-zerop (rewrite)
             (implies
              (not (lessp 0 c))
              (zerop c)))

(prove-lemma fix-times (rewrite)
             (equal
              (fix (times x y))
              (times x y)))
```

```
(prove-lemma lessp-rplus-difference-bridge (rewrite)
             (implies
              (lessp 0 (numerator (fix-rational z)))
              (rlessp (rdifference x z) x))
             ((enable rplus simple-rplus rlessp-reduce1 itimes iplus
                      rlessp-reduce2 negativep-numerator-reduce fix-rational implies
                      *1*lessp and or integerp fix-int rlessp zerop times-zero-2
                      negative-guts-minus ilessp rneg ineg simple-rneg
                      denominator-0 negativep-means-not-lessp lessp-difference-rewrite
                      lessp-0-from-not-zerop not-lessp-0-means-zerop
                      negative-guts-numerator-reduce-x-0 lessp-times-0-rewrite fix-times)
              (enable-theory r2 arithmetic rational-defns)
              (disable-theory t)))

(prove-lemma find-func-lessp-fact1
             (rewrite)
             (implies (and (lessp 0 (numerator (fix-rational z)))
                           (not (rlessp (rdifference x z) a)))
                      (lessp (find-func-zero-measure a
                                                     (rdifference x z)
                                                     z)
                             (find-func-zero-measure a x z)))
             )

(prove-lemma rlessp-rdifference
             (rewrite)
             (and (equal (rlessp (rdifference x y) z)
                         (rlessp x (rplus y z)))
                  (equal (rlessp z (rdifference x y))
                         (rlessp (rplus y z) x)))
             )


(prove-lemma find-func-lessp-fact2 (rewrite)
             (implies
              (and
               (lessp 0 (numerator (fix-rational z)))
               (not (rlessp a (rplus x z))))
             (lessp (find-func-zero-measure (rplus x z) a z)
                    (find-func-zero-measure x a z)))
             ((enable-theory r2 arithmetic)
              (disable-theory t)
              (disable rdifference)
              (enable rquotient find-func-zero-measure rlessp-rdifference)
              (use (find-func-lessp-fact1 (x a) (a x) (z z)))))


(prove-lemma quotient-zerop-arg1 (rewrite)
             (implies
              (zerop x)
              (equal (quotient x y) 0)))


(prove-lemma times-quotient-remainder-fact (rewrite)
             (implies
              (equal (remainder b a) 0)
             (equal
              (times a (quotient b a))
              (fix b))))

(prove-lemma quotient-reduce (rewrite)
             (equal
              (quotient (numerator (reduce x))
                        (denominator (reduce x)))
              (quotient (numerator (fix-rational x)) (denominator (fix-rational x))))
             ((enable-theory r2)
              (enable reduce fix-rational)))

(prove-lemma floor-reduce (rewrite)
             (equal
              (floor (reduce x))
              (floor x))
             ((enable-theory r2)
              (enable numberp-numerator-reduce)))

(prove-lemma plus-zerop-2 (rewrite)
             (implies
              (zerop x)
             (equal
              (plus y x)
              (fix y))))

(prove-lemma quotient-preserves-lessp (rewrite)
             (implies
              (not (lessp b a))
              (equal (lessp (quotient b c) (quotient a c)) f))
             ((induct (double-remainder-induction a b c))))


(prove-lemma not-lessp-times-means-not-lessp-quotient
        (rewrite)
        (implies (and (not (lessp (times dy nx) (times dx ny)))
                      (not (zerop dy))
                      (not (zerop dx)))
                 (not (lessp (quotient nx dx)
                             (quotient ny dy))))
        )



(prove-lemma not-rlessp-neg-pos (rewrite)
             (implies
              (and
               (not (rlessp x y))
               (not (numberp (numerator x)))
               (numberp (numerator y)))
              (and
               (not (rationalp x))
               (implies (rationalp y) (equal (numerator y) 0))))
             ((enable-theory r2)
              (enable ilessp)))


(prove-lemma rlessp-means-not-lessp-times-bridge (rewrite)
             (implies
              (and
               (rationalp x)
               (rationalp y)
               (numberp (numerator y))
               (numberp (numerator x))
               (not (rlessp x y)))
              (equal (lessp (quotient (numerator x) (denominator x))
                            (quotient (numerator y) (denominator y)))
                     f))
             ((enable rlessp itimes ilessp)
              (enable-theory r2)
              (disable not-lessp-times-means-not-lessp-quotient)
              (use (not-lessp-times-means-not-lessp-quotient
                    (nx (numerator x)) (dx (denominator x))
                    (ny (numerator y)) (dy (denominator y))))))
```

```
(prove-lemma lessp-floor (rewrite)
             (implies
              (not (rlessp x y))
              (equal (lessp (floor x) (floor y)) f))
             ((enable ilessp)
              (enable-theory r2)))

(disable rlessp-means-not-lessp-times-bridge)

(prove-lemma rationalp-fpminspace (rewrite)
             (rationalp (fpminspace))
             ((use (rlessp-0-fpminspace))
              (disable rlessp-0-fpminspace)))

(prove-lemma numerator-fpminspace-0 (rewrite)
             (equal
              (equal (numerator (fpminspace)) 0)
              f)
             ((use (rlessp-0-fpminspace))
              (enable-theory r2)
              (enable ilessp)
              (disable rlessp-0-fpminspace)))

(prove-lemma numberp-numerator-fpminspace (rewrite)
             (numberp (numerator (fpminspace)))
             ((use (rlessp-0-fpminspace))
              (disable rlessp-0-fpminspace)))

(prove-lemma numerator-simple-rplus-x-simple-rneg-y-0 (rewrite)
             (equal
              (equal (numerator (simple-rplus y (simple-rneg x))) 0)
              (requal x y))
             ((enable-theory r2 arithmetic rational-defns integers)
              (enable rlessp ilessp itimes rplus simple-rplus rneg simple-rneg ineg
                      fix-int iplus *1*fix-rational numberp-numerator-fix-rational
                      integerp-minus implies numberp-numerator-reduce
                      negativep-numerator negativep-numerator-reduce
                      negativep-numerator-fix-rational denominator-0 requal
                      negative-guts-minus integerp zerop and minus-equal
                      correctness-of-cancel-lessp-times not lessp)
              (disable-theory t)))

(prove-lemma quotient-gcd-fact (rewrite)
             (implies
              (not (zerop x))
              (and
               (lessp 0 (quotient x (gcd x y)))
               (lessp 0 (quotient x (gcd y x))))))

(prove-lemma lessp-times-quotient-gcd-bridge
             (rewrite)
             (and (equal (lessp (times a (quotient b (gcd b d)))
                                (times c (quotient d (gcd b d))))
                         (lessp (times a b) (times c d)))
                  (equal (lessp (times a (quotient b (gcd d b)))
                                (times c (quotient d (gcd d b))))
                         (lessp (times a b) (times c d))))
             )

(prove-lemma numerator-simple-rplus-reduce-0 (rewrite)
             (equal
              (equal (numerator (simple-rplus x (reduce y))) 0)
              (equal (numerator (simple-rplus x y)) 0))
             ((enable-theory r2 rational-defns arithmetic)
              (disable-theory t)
              (enable reduce simple-rplus iplus itimes fix-int integerp-minus zerop
                      negativep-numerator-fix-rational nrationalp-fix-rational implies
                      numberp-numerator-fix-rational implies and not quotient-gcd-fact
                      lessp-0-from-not-zerop lessp and not times-quotient-gcd-bridge
                      fix-rational-of-rationalp integerp integerp-minus
                      times-quotient-gcd-bridge lessp-times-quotient-gcd-bridge
                      negative-guts-minus times-zero-2 fix-rational)))

(prove-lemma numerator-rplus-x-rneg-y-0 (rewrite)
             (equal
              (equal (numerator (rplus y (rneg x))) 0)
              (requal x y))
             ((enable rplus rneg rationalp-simple-rplus)))

(prove-lemma rlessp-1 (rewrite)
             (implies
              (lessp a b)
              (rlessp (rational a b) (rational 1 1)))
             ((enable-theory r2)
              (enable rlessp ilessp itimes iplus fix-int fix-rational)))

(prove-lemma numberp-numerator-rplus (rewrite)
             (implies
              (and
               (numberp (numerator (fix-rational x)))
               (numberp (numerator (fix-rational y))))
              (numberp (numerator (rplus x y)))))

(prove-lemma lessp-plus-times-bridge
             (rewrite)
             (implies (and (numberp c)
                           (numberp x1)
                           (lessp x1 d)
                           (lessp w z)
                           (not (equal z 0))
                           (numberp z)
                           (not (lessp v c)))
                      (equal
                       (lessp (plus (times x1 z) (times c d z))
                              (plus (times d w)
                                    (times d z)
                                    (times d v z)))
                       t))
             )
```

```
(prove-lemma lessp-floor-t-bridge (rewrite)
             (implies (and (numberp (numerator a))
                      (not (rlessp b (rplus (rational 1 1) a)))
                      (rationalp b)
                      (numberp (numerator b))
                      (rationalp a))
                      (equal (lessp (quotient (numerator a)
                                              (denominator a))
                                    (quotient (numerator b)
                                              (denominator b)))
                             t))
             ((enable-theory r2 arithmetic rational-defns)
              (disable-theory t)
              (enable rlessp simple-rplus rplus iplus itimes fix-int integerp
                      ilessp rlessp-reduce2 not implies and zerop
                      lessp-plus-times-bridge)))


(prove-lemma rlessp-rplus-rneg (rewrite)
             (and
              (equal
               (rlessp (rplus (rneg x) y) z)
               (rlessp y (rplus x z)))
              (equal
               (rlessp (rplus y (rneg x)) z)
               (rlessp y (rplus x z)))
              (equal
               (rlessp z (rplus (rneg x) y))
               (rlessp (rplus x z) y))
              (equal
               (rlessp z (rplus y (rneg x)))
               (rlessp (rplus x z) y)))
             ((use (rlessp-rplus-rplus (c x) (x (rplus (rneg x) y)) (y z))
                   (rlessp-rplus-rplus (c x) (y (rplus (rneg x) y)) (x z)))
              (disable rlessp-rplus-rplus)
              (disable-theory t)
              (enable rlessp-reduce1 rlessp-reduce2 implies and)
              (enable-theory r2)))


(prove-lemma rlessp-rneg (rewrite)
             (and
              (equal
               (rlessp (rneg x) y)
               (rlessp (rational 0 1) (rplus x y)))
              (equal
               (rlessp x (rneg y))
               (rlessp (rplus x y) (rational 0 1))))
             ((use (rlessp-rplus-rplus (c x) (x (rneg x)) (y y))
                   (rlessp-rplus-rplus (c y) (x x) (y (rneg y))))
              (disable rlessp-rplus-rplus)
              (enable-theory r2)
              (disable-theory t)
              (enable rlessp-reduce1 rlessp-reduce2 implies and)))


;
;; redo this so that it fires first
(prove-lemma rlessp-rplus-x-x (rewrite)
             (equal
              (rlessp (rplus x y) (rplus x z))
              (rlessp y z)))

(prove-lemma lessp-floor-t (rewrite)
             (implies
              (and
               (numberp (numerator (fix-rational a)))
               (not (rlessp b (rplus a (rational 1 1)))))
              (equal (lessp (floor a) (floor b)) t))
             ((enable-theory r2)
              (enable ilessp)))


(prove-lemma rlessp-rtimes-x-x (rewrite)
             (equal
              (rlessp (rtimes a b) (rtimes a c))
              (if (rzerop a)
                  f
                (if (numberp (numerator (fix-rational a)))
                    (rlessp b c)
                  (rlessp c b))))
             ((enable-theory r2 arithmetic rational-defns)
              (disable-theory t)
              (enable rlessp rtimes fix-rational itimes ilessp iplus simple-rtimes
                      rlessp-reduce1 rlessp-reduce2 and implies not times-zero-2
                      fix-int integerp-minus lessp zerop integerp lessp-0-from-not-zerop
                      numberp-numerator-fix-rational negative-guts-minus
                      correctness-of-cancel-lessp-times)))


(disable rlessp-rplus-rneg)

(defn move-rlessp-args-right (x)
  (if (equal (car x) 'rlessp)
      (if (not (equal (cadr x) (list 'rational ''0 ''1)))
          (list 'rlessp (list 'rational ''0 ''1)
                (list 'rplus (list 'rneg (cadr x)) (caddr x)))
        x)
    x))


(prove-lemma eval$-rneg (rewrite)
             (implies
              (equal (car x) 'rneg)
              (equal (eval$ t x a)
                     (rneg (eval$ t (cadr x) a)))))

(prove-lemma eval$-rlessp (rewrite)
             (implies
              (equal (car x) 'rlessp)
              (equal (eval$ t x a)
                     (rlessp (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))

(prove-lemma eval$-rational (rewrite)
             (implies
              (equal (car x) 'rational)
              (equal (eval$ t x a)
                     (rational (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))

(prove-lemma move-rlessp-args-right-correct ((meta rlessp))
             (equal (eval$ t x a)
                    (eval$ t (move-rlessp-args-right x) a))
             ((use (rlessp-rplus-x-x (x (rneg (eval$ t (cadr x) a)))
                                     (y (eval$ t (cadr x) a))
                                     (z (eval$ t (caddr x) a))))
              (enable-theory r2)
              (enable eval$-rplus eval$-rneg eval$-rational eval$-rlessp)
              (disable rlessp-rplus-x-x rlessp-rplus-rplus)))
```

```
(disable move-rlessp-args-right-correct)


(prove-lemma lessp-floor-t-better
             (rewrite)
             (implies (and (not (rlessp b (rational 1 1)))
                           (not (rlessp b (rplus a (rational 1 1)))))
                      (equal (lessp (floor a) (floor b)) t))
             )

(lemma lessp-floor-rplus (rewrite)
       (implies
        (not (rlessp x y))
        (and
         (equal (lessp (floor (rplus c x)) (floor (rplus c y))) f)
         (equal (lessp (floor (rplus c x)) (floor (rplus y c))) f)
         (equal (lessp (floor (rplus x c)) (floor (rplus c y))) f)
         (equal (lessp (floor (rplus x c)) (floor (rplus y c))) f)))
       ((enable-theory r2)
        (enable lessp-floor rlessp-rplus-x-x)))

(lemma rtimes-monotonic (rewrite)
       (implies
        (and
         (numberp (numerator a))
         (not (rlessp x y)))
        (and
         (not (rlessp (rtimes a x) (rtimes a y)))
         (not (rlessp (rtimes a x) (rtimes y a)))
         (not (rlessp (rtimes x a) (rtimes a y)))
         (not (rlessp (rtimes x a) (rtimes y a)))))
       ((enable-theory r2)
        (enable rlessp-rtimes-x-x fix-rational)))

(lemma rlessp-rneg-rneg (rewrite)
       (equal
        (rlessp (rneg x) (rneg y))
        (rlessp y x))
       ((enable-theory r2 arithmetic rational-defns)
        (enable rlessp ilessp rneg simple-rneg ineg fix-rational itimes
                rlessp-reduce1 rlessp-reduce2 integerp fix-int)))

(lemma find-func-zero-measure-monotonic (rewrite)
       (implies
        (and
         (not (rlessp x y))
         (numberp (numerator b)))
        (and
         (equal (lessp (find-func-zero-measure a x b)
                       (find-func-zero-measure a y b)) f)
         (equal (lessp (find-func-zero-measure y a b)
                       (find-func-zero-measure x a b)) f)))
       ((enable-theory r2)
        (enable lessp-floor-rplus find-func-zero-measure rtimes-monotonic
                rquotient numberp-numerator-rinverse rlessp-rneg-rneg
                numberp-numerator-fix-rational)))

(lemma rlessp-means-not-requal (rewrite)
       (implies
        (rlessp a b)
        (not (requal a b)))
       ((enable-theory r2 arithmetic rational-defns)
        (enable rlessp ilessp rneg simple-rneg ineg fix-rational itimes requal
                rlessp-reduce1 rlessp-reduce2 integerp fix-int)))

(prove-lemma negative-guts-numerator-0 (rewrite)
             (implies
              (rationalp x)
              (equal
               (equal (negative-guts (numerator x)) 0)
               (numberp (numerator x))))
              ((enable-theory r2) (enable rzerop)))


(prove-lemma numberp-numerator-rplus-x-rneg-y-better
             (rewrite)
             (equal (numberp (numerator (rplus x (rneg y))))
                    (not (rlessp x y)))
             )


(prove-lemma lessp-floor-bridge1 (rewrite)
             (implies
              (and
               (not (rlessp x y))
               (equal (remainder (numerator x) (denominator x)) 0)
               (equal (remainder (numerator y) (denominator y)) 0))
              (equal (lessp (floor x) (floor y)) f))
             ((enable rlessp ilessp fix-int itimes floor implies lessp
                      negativep-numerator-fix-rational nrationalp-fix-rational
                      quotient-zerop-arg1 zerop rationalp-zerop rationalp-non-integer
                      not and integerp correctness-of-cancel-lessp-times fix)
              (enable-theory r2 arithmetic rational-defns)
              (disable-theory t)))


(lemma find-func-zero-measure-reduce (rewrite)
       (and
        (equal
         (find-func-zero-measure (reduce a) b c)
         (find-func-zero-measure a b c))
        (equal
         (find-func-zero-measure a (reduce b) c)
         (find-func-zero-measure a b c))
        (equal
         (find-func-zero-measure a b (reduce c))
         (find-func-zero-measure a b c)))
       ((enable-theory r2) (enable find-func-zero-measure rquotient
                                   rinverse-reduce)))

(prove-lemma fpp-means-find-func-ok
 (rewrite)
 (implies (and (fpp x)
               (fpp m)
               (fpp y)
               (rlessp x m)
               (rlessp m y))
          (and (lessp (find-func-zero-measure x m
                                              (fpminspace))
                      (find-func-zero-measure x y
                                              (fpminspace)))
               (lessp (find-func-zero-measure m y
                                              (fpminspace))
                      (find-func-zero-measure x y
                                              (fpminspace)))))
 )
```

```
(lemma fpp-rationalp (rewrite)  ; was axiom - changed 3-26-90
       (implies
        (fpp x)
        (rationalp x))
       ((enable fpp-round-intro)))

(prove-lemma rationalp-round (rewrite)
             (rationalp (round x)))

(prove-lemma round-0 (rewrite)
             (implies
              (and
               (fpp x)
               (equal (numerator x) 0))
              (equal (numerator (round x)) 0))
             ((use (not-round-down-past (y (rational 0 1)) (x x))
                   (not-round-up-past (y (rational 0 1)) (x x)))
              (enable-theory r2)
              (enable rationalp ilessp)))

(prove-lemma fix-rational-round (rewrite)
             (equal
              (fix-rational (round x))
              (round x))
             ((enable fix-rational)))

(lemma rtimes-rinverse-rinverse (rewrite)
       (equal (rtimes (rinverse x) (rinverse y))
              (rinverse (rtimes x y)))
       ((enable-theory r2 rational-defns arithmetic)
        (enable rinverse rtimes simple-rinverse simple-rtimes reduce itimes
                fix-int ineg equal-reduce-reduce requal
                requal-simple-rtimes-bridge
                requal-simple-rinverse-reduce-bridge integerp-minus integerp
                fix-rational numerator-fix-rational-0 rationalp-0)))

(lemma reduce-rinverse (rewrite)
       (equal
        (reduce (rinverse x))
        (rinverse x))
       ((enable rinverse)
        (enable-theory r2)))

(prove-lemma rtimes-rinverse-hack
             (rewrite)
             (and (equal (rtimes a (rinverse (rtimes a b)))
                         (if (rzerop a)
                             (rational 0 1)
                             (rinverse b)))
                  (equal (rtimes a (rinverse (rtimes b a)))
                         (if (rzerop a)
                             (rational 0 1)
                             (rinverse b))))
             )

(prove-lemma rtimes-rinverse-hack2
             (rewrite)
             (and (equal (rtimes b
                                 (rtimes (rinverse (rtimes a b)) c))
                         (if (rzerop b)
                             (rational 0 1)
                             (rtimes (rinverse a) c)))
                  (equal (rtimes b
                                 (rtimes (rinverse (rtimes b a)) c))
                         (if (rzerop b)
                             (rational 0 1)
                             (rtimes (rinverse a) c))))
             )

(prove-lemma rtimes-rinverse-hack3
             (rewrite)
             (and (equal (rtimes a (rtimes (rinverse a) b))
                         (if (rzerop a)
                             (rational 0 1)
                             (reduce b)))
                  (equal (rtimes a (rtimes b (rinverse a)))
                         (if (rzerop a)
                             (rational 0 1)
                             (reduce b))))
             )

(prove-lemma reduce-rzerop (rewrite)
             (implies
              (rzerop x)
             (equal (reduce x) (rational 0 1)))
             ((enable reduce rzerop)))

(prove-lemma equal-numerator-rinverse-0 (rewrite)
             (equal
              (equal (numerator (rinverse x)) 0)
              (rzerop x))
             ((enable rzerop rinverse simple-rinverse ineg)
              (enable-theory r2)))

(prove-lemma rinverse-0 (rewrite)
             (implies
              (rzerop x)
              (equal (rinverse x) (rational 0 1)))
             ((enable rinverse simple-rinverse reduce-0)))

(prove-lemma rzerop-rneg (rewrite)
             (equal
              (rzerop (rneg x))
              (rzerop x))
             ((enable-theory r2) (enable rzerop rneg simple-rneg ineg)))

(prove-lemma rzerop-rinverse (rewrite)
             (equal
              (rzerop (rinverse x))
              (rzerop x))
             ((enable-theory r2)
              (enable rzerop rinverse simple-rinverse ineg)))
```

```
(lemma rzerop-rtimes-rewrite (rewrite)
             (equal
              (rzerop (rtimes a b))
              (or
               (rzerop a)
               (rzerop b)))
             ((enable-theory r2 rational-defns arithmetic integers)
              (enable rzerop rtimes simple-rtimes itimes fix-int-on-integers
                      integerp-minus nrationalp-fix-rational reduce-0
                      *1*fix-int negativep-numerator-fix-rational integerp
                      rationalp-not-rational-formp numerator-reduce-0)))


(prove-lemma rlessp-rinverse-hack
             (rewrite)
             (implies (and (equal (numberp (numerator x))
                                  (numberp (numerator y)))
                           (not (rzerop x))
                           (not (rzerop y)))
                      (equal (rlessp (rinverse x) y)
                             (rlessp (rinverse y) x)))
             )


(lemma round-min-bounds (rewrite)
       (and
        (not (rlessp (rational 1 1) (round-min)))
        (rlessp (rational 99 100) (round-min)))
       ((enable fpp-round-intro)))

(lemma round-max-bounds (rewrite)
       (and
        (not (rlessp (round-max) (rational 1 1)))
        (rlessp (round-max) (rational 101 100)))
       ((enable fpp-round-intro)))

(prove-lemma rzerop-round-max (rewrite)
             (not (rzerop (round-max)))
             ((use (round-max-bounds))
              (enable rzerop)
              (disable round-max-bounds)))

(lemma lessp-times-3 (rewrite)
       (implies
        (and
         (not (lessp a c))
         (not (lessp b d)))
        (equal (lessp (times a b) (times c d)) f)))

(prove-lemma rlessp-rtimes-simple-hack
             (rewrite)
             (implies (and (numberp (numerator (fix-rational a)))
                           (numberp (numerator (fix-rational b)))
                           (numberp (numerator (fix-rational c)))
                           (numberp (numerator (fix-rational d)))
                           (not (rlessp c a))
                           (not (rlessp d b)))
                      (not (rlessp (rtimes c d) (rtimes a b))))
             )

(prove-lemma rlessp-rtimes-bound
             (rewrite)
             (implies (and (numberp (numerator a))
                           (numberp (numerator b))
                           (not (rlessp c a))
                           (not (rlessp d b))
                           (not (rlessp e (rtimes c d))))
                      (not (rlessp e (rtimes a b))))
             )


(prove-lemma numberp-numerator-round-max (rewrite)
             (numberp (numerator (round-max)))
             ((use (round-max-bounds))
              (disable round-max-bounds)
              (enable rlessp ilessp itimes)
              (enable-theory r2)))

(prove-lemma rationalp-round-min (rewrite)
             (rationalp (round-min))
             ((use (round-min-bounds))
              (disable round-min-bounds)))

(lemma numberp-numerator-round-min (rewrite)
       (numberp (numerator (round-min)))
       ((use (round-min-bounds))
        (disable round-min-bounds times-add1)
        (enable rlessp ilessp itimes numberp-numerator-fix-rational
                negativep-numerator-fix-rational *1*fix-rational
                fix-int-numerator fix-int-on-integers integerp-minus
                rationalp-round-min fix-rational-of-rationalp
                negative-guts-numerator-0
                lessp-times-0-rewrite)
        (enable-theory r2 arithmetic rational-defns)))

(prove-lemma round-max-hack
  (rewrite)
  (implies (numberp (numerator a))
           (not (rlessp (rtimes (rinverse (rtimes (round-max) (round-max)))
                                (rtimes (rational 2 1) a))
                        a)))
  )

(PROVE-LEMMA ROUND-MIN-HACK
             (REWRITE)
             (NUMBERP (NUMERATOR (RPLUS (RNEG (RTIMES (ROUND-MIN) (ROUND-MIN)))
                                        (RATIONAL 2 1))))
             )
```

```
(constrain mid-bound1-intro (rewrite)
           (and
            (fpp (mid-bound1))
            (implies
             (and
              (fpp x)
              (numberp (numerator x))
              (not (rlessp (fpmaximum) (rtimes (rational 2 1) x))))
             (and
              (not (rlessp (rquotient (rtimes (rdifference (rational 2 1)
                                                   (rtimes (round-max)
                                                           (round-max)))
                                              x)
                                      (rtimes (round-max) (round-max)))
                           (round (rtimes (mid-bound1) x))))

              (not (rlessp (rquotient (rtimes (rtimes (round-min) (round-min))
                                              x)
                                      (rdifference (rational 2 1)
                                                   (rtimes (round-min)
                                                           (round-min))))
                           (round (rtimes (mid-bound1) x)))))))
           ((mid-bound1 (lambda nil (rational 0 1))))
           ((enable-theory r2 ground-zero rational-defns)
            (disable-theory t)
            (enable rquotient rzerop-rtimes fpp-0 round-0 rlessp-0
                    fix-rational-round negativep-numerator
                    numberp-numerator-rplus-x-rneg-y-better reduce-nrationalp
                    rtimes-rinverse-hack2 rlessp-x-x rtimes-rinverse-hack
                    rtimes-rinverse-hack3 reduce-rzerop rlessp-reduce1
                    rlessp-reduce2 nrationalp-rlessp *1*rinverse
                    numberp-numerator-rtimes numberp-numerator-rinverse
                    numberp-numerator-round-min round-min-hack
                    rationalp-not-rational-formp round-max-hack)
            (disable rtimes-rinverse-rinverse)))



(prove-lemma numberp-numerator-round (rewrite)
             (implies
              (and
               (rationalp x)
               (numberp (numerator x)))
              (numberp (numerator (round x))))
             ((use (not-round-down-past (y (rational 0 1)) (x x)))
              (disable not-round-down-past)
              (enable-theory r2)))

(lemma rlessp-rlessp-requal (rewrite)
       (implies
        (and
         (not (rlessp a b))
         (not (rlessp b a)))
        (requal a b))
       ((enable rlessp ilessp requal itimes)
        (enable-theory arithmetic)))

(prove-lemma round-of-small-fact
             (rewrite)
             (implies (and (rationalp x)
                           (numberp (numerator x))
                           (not (rlessp (fpminimum) x)))
                      (or (rzerop (round x))
                          (requal (round x) (fpminimum))))
             )
```

```
(prove-lemma rlessp-rplus-fact1
             (rewrite)
             (equal (rlessp (rplus a b) b)
                    (negativep (numerator (fix-rational a))))
             )

(lemma rlessp-rplus-fact2 (rewrite)
       (equal
        (rlessp (rplus b a) b)
        (negativep (numerator (fix-rational a))))
       ((enable-theory r2)
        (enable rlessp-rplus-fact1)))

(prove-lemma rlessp-rplus-pair-fact
             (rewrite)
             (implies (and (not (rlessp a b))
                           (not (rlessp c d)))
                      (not (rlessp (rplus a c) (rplus b d))))
             )

(prove-lemma rationalp-round-max (rewrite)
             (rationalp (round-max))
             ((use (round-max-bounds))
              (disable round-max-bounds)))

(lemma rationalp-fpmaximum (rewrite)
       (rationalp (fpmaximum))
       ((enable fpp-rationalp fpp-maximum)))

(lemma fpmaximum-bound (rewrite)
       (not (rlessp (fpmaximum) (rational 1 1)))
       ((enable fpp-round-intro)))

(lemma fix-rational-fpmaximum (rewrite)
       (equal (fix-rational (fpmaximum))
              (fpmaximum))
       ((enable fix-rational-of-rationalp rationalp-fpmaximum)))

(prove-lemma numberp-numerator-fpmaximum (rewrite)
             (numberp (numerator (fpmaximum)))
             ((use (fpmaximum-bound))
              (disable fpmaximum-bound)))

(prove-lemma numberp-numerator-fpminimum (rewrite)
             (numberp (numerator (fpminimum)))
             ((enable fpp-round-intro)))

(prove-lemma middle-fact1
 (rewrite)
 (implies (and (numberp (numerator a))
               (numberp (numerator b))
               (not (rlessp (fpmaximum)
                            (rtimes a (rational 2 1))))
               (not (rlessp (fpmaximum)
                            (rtimes b (rational 2 1))))
               (rlessp a
                       (round (rtimes b (mid-bound1))))
               (fpp a)
               (fpp b)
               (rlessp (fpminimum) b))
          (rlessp (round (rquotient (round (rplus a b))
                                    (rational 2 1)))
                  b))
 )
```

```
(lemma rtimes-2-x (rewrite)
       (and
        (equal
         (rtimes (rational 2 1) x)
         (rplus x x))
        (equal
         (rtimes x (rational 2 1))
         (rplus x x)))
       ((enable-theory r2 arithmetic integers rational-defns)
        (enable rplus simple-rplus equal-reduce-reduce requal itimes
                fix-int-on-integers integerp-minus rtimes simple-rtimes
                iplus fix-rational *1*fix-int *1*rationalp
                integerp-if-numberp)))

(lemma rlessp-rtimes-cancel (rewrite)
       (implies
        (and
         (numberp (numerator (fix-rational a)))
         (numberp (numerator (fix-rational b))))
        (and
         (equal
          (rlessp (rtimes a b) b)
          (and (rlessp a (rational 1 1))
               (not (rzerop b))))
         (equal
          (rlessp (rtimes b a) b)
          (and (rlessp a (rational 1 1))
               (not (rzerop b))))
         (equal
          (rlessp b (rtimes a b))
          (and (rlessp (rational 1 1) a)
               (not (rzerop b))))
         (equal
          (rlessp b (rtimes b a))
          (and (rlessp (rational 1 1) a)
               (not (rzerop b))))))

       ((enable-theory r2 arithmetic integers rational-defns)
        (enable rplus simple-rplus equal-reduce-reduce requal itimes
                fix-int-on-integers integerp-minus rtimes simple-rtimes
                iplus fix-rational *1*fix-int *1*rationalp rlessp
                rlessp-reduce1 *1*ilessp ilessp
                rlessp-reduce2 nrationalp-rlessp *1*rlessp rzerop rlessp-0
                integerp-if-numberp correctness-of-cancel-lessp-times)))

(prove-lemma bound-fact
 (rewrite)
 (implies (and (not (rlessp b a))
               (not (rlessp (fpmaximum) (rplus a b)))
               (not (rlessp a
                            (rquotient (fpminimum) (round-min)))))
          (not (rlessp (rquotient (round (rplus a b))
                                  (rational 2 1))
                       (fpminimum))))
 )

(lemma rlessp-round-fpmaximum (rewrite)
       (implies
        (and
         (numberp (numerator (fix-rational x)))
         (rationalp x))
        (not (rlessp (fpmaximum) (round x))))
       ((enable fpp-round rmagnitude-positive numberp-numerator-round
                fix-rational-of-rationalp fpp-rationalp rlessp-reduce2)
        (use (fpp-bounded-fpmaximum (x (round x))))))

(prove-lemma equal-numerator-round-min-0 (rewrite)
       (not (equal (numerator (round-min)) 0))
       ((use (round-min-bounds))
        (enable rlessp-0)
        (disable round-min-bounds)
        (enable-theory r2 rational-defns)))

(prove-lemma negative-guts-returns-non-zero (rewrite)
             (implies
              (and
               (integerp x)
               (not (numberp x)))
              (lessp 0 (negative-guts x))))

(prove-lemma lessp-times-times (rewrite)
             (implies
              (and
               (lessp a b)
               (lessp c d))
              (lessp (times a c) (times b d))))

(prove-lemma lessp-times-times2 (rewrite)
             (implies
              (and
               (not (lessp a b))
               (not (lessp c d)))
              (not (lessp (times a c) (times b d)))))

(lemma lessp-times-a-a-times-b-b (rewrite)
       (equal
        (lessp (times a a) (times b b))
        (lessp a b))
       ((use (lessp-times-times2 (a a) (b b) (c a) (d b))
             (lessp-times-times  (a a) (b b) (c a) (d b)))
        (enable-theory arithmetic)
        (disable times-sub1)))

(lemma rlessp-rtimes-a-a-1 (rewrite)
       (and
        (equal
         (rlessp (rtimes a a) (rational 1 1))
         (rlessp (rmagnitude a) (rational 1 1)))
        (equal
         (rlessp (rational 1 1) (rtimes a a))
         (rlessp (rational 1 1) (rmagnitude a))))
       ((disable times-add1)
        (enable-theory r2 arithmetic integers rational-defns)
        (enable rplus simple-rplus equal-reduce-reduce requal itimes
                fix-int-on-integers integerp-minus rtimes simple-rtimes
                iplus fix-rational *1*fix-int *1*rationalp rlessp
                rlessp-reduce1 *1*ilessp ilessp rmagnitude
                simple-rmagnitude rneg simple-rneg
                ineg lessp-times-a-a-times-b-b
                rlessp-reduce2 nrationalp-rlessp *1*rlessp rzerop
                integerp-if-numberp correctness-of-cancel-lessp-times)))

(prove-lemma round-min-2-hack
         (rewrite)
         (rlessp (rtimes (round-min) (round-min))
                 (rational 2 1))
         )

(lemma rzerop-round-min (rewrite)
       (not (rzerop (round-min)))
       ((enable-theory r2)
        (enable rzerop rationalp-round-min equal-numerator-round-min-0)
        (use (round-min-bounds))))
```

```
(prove-lemma middle-fact2
 (rewrite)
 (implies (and (numberp (numerator a))
               (numberp (numerator b))
               (not (rlessp (fpmaximum)
                            (rtimes a (rational 2 1))))
               (not (rlessp (fpmaximum)
                            (rtimes b (rational 2 1))))
               (rlessp a
                       (round (rtimes b (mid-bound1))))
               (fpp a)
               (fpp b)
               (not (rlessp a
                            (rquotient (fpminimum) (round-min))))
               (not (rlessp b a)))
          (rlessp a
                  (round (rquotient (round (rplus a b))
                                    (rational 2 1)))))
 )

(constrain mid-bound2-intro (rewrite)
           (and (fpp (mid-bound2))
                (or (rlessp (fpmaximum) (rquotient (fpminimum) (round-min)))
                    (not (rlessp (mid-bound2)
                                 (rquotient (fpminimum) (round-min))))))
           ((mid-bound2 (lambda () (fpmaximum)))))

(prove-lemma numberp-numerator-mid-bound2
 (rewrite)
 (implies (not (rlessp (fpmaximum)
                       (rquotient (fpminimum) (round-min))))
          (numberp (numerator (mid-bound2))))
 )


(constrain mid-bound3-intro (rewrite)
           (and (fpp (mid-bound3))
                (not (rlessp (rquotient (fpmaximum) (rational 2 1))
                             (mid-bound3))))
           ((mid-bound3 (lambda () (rational 0 1))))
           ((enable rquotient)))

(defn fp-mid (x y)
  (if (and (rlessp x (mid-bound2))
           (rlessp (mid-bound2) y))
      (mid-bound2)
    (if (and (rlessp x (rational 0 1))
             (rlessp (rational 0 1) y))
        (rational 0 1)
      (if (and (rlessp x (rneg (mid-bound2)))
               (rlessp (rneg (mid-bound2)) y))
          (rneg (mid-bound2))
        (round (rquotient (round (rplus x y))
                          (rational 2 1)))))))

(lemma fpp-mid (rewrite)
       (fpp (fp-mid x y))
       ((enable fp-mid fpp-round-intro mid-bound2-intro)))

(prove-lemma rlessp-mid-bound3-hack
          (rewrite)
          (implies (not (rlessp (mid-bound3) x))
                   (not (rlessp (fpmaximum)
                                (rtimes x (rational 2 1)))))
          )

(prove-lemma fp-mid-fact1
 (rewrite)
 (implies (and (numberp (numerator a))
               (rlessp a b)
               (not (rlessp (mid-bound3) (rmagnitude a)))
               (not (rlessp (mid-bound3) (rmagnitude b)))
               (fpp a)
               (fpp b)
               (rlessp (mid-bound2) b)
               (rlessp a
                       (round (rtimes b (mid-bound1))))
               (not (rlessp (fpmaximum)
                            (rquotient (fpminimum) (round-min)))))
          (and (rlessp (fp-mid a b) b)
               (rlessp a (fp-mid a b))))
 )

(prove-lemma fix-rational-rzerop (rewrite)
             (implies
              (rzerop x)
              (equal (numerator (fix-rational x)) 0))
             ((enable rzerop reduce-0 fix-rational)))

(prove-lemma rmagnitude-rneg  (rewrite)
             (equal (rmagnitude (rneg x))
                    (rmagnitude x))
             )

(prove-lemma fp-mid-hack1
             (rewrite)
             (implies (rlessp (fp-mid (rneg b) (rneg a))
                              (rneg a))
                      (rlessp a (fp-mid a b)))
             )

(prove-lemma fp-mid-hack2
             (rewrite)
             (implies (rlessp (rneg b)
                              (fp-mid (rneg b) (rneg a)))
                      (rlessp (fp-mid a b) b))

            )
(prove-lemma fp-mid-fact2
             (rewrite)
             (implies (and (not (numberp (numerator a)))
                           (or (rzerop b)
                               (negativep (numerator b)))
                           (rlessp a b)
                           (not (rlessp (mid-bound3) (rmagnitude a)))
                           (not (rlessp (mid-bound3) (rmagnitude b)))
                           (fpp a)
                           (fpp b)
                           (not (rlessp (fpmaximum)
                                        (rquotient (fpminimum) (round-min))))
                           (rlessp (round (rtimes a (mid-bound1)))
                                   b)
                           (rlessp (mid-bound2) (rneg a)))
                      (and (rlessp (fp-mid a b) b)
                           (rlessp a (fp-mid a b))))
             )
```

```
(lemma fpp-means-find-func-ok-rewrite  (rewrite)
 (implies (and (fpp x)
               (fpp m)
               (fpp y)
               (rlessp x m)
               (rlessp m y))
          (and (equal (lessp (find-func-zero-measure x m
                                                        (fpminspace))
                             (find-func-zero-measure x y
                                                        (fpminspace)))
                      t)
               (equal (lessp (find-func-zero-measure m y
                                                        (fpminspace))
                             (find-func-zero-measure x y
                                                        (fpminspace)))
                      t)))
 ((use (fpp-means-find-func-ok))))


(prove-lemma fp-mid-fact3
 (rewrite)
 (implies (and (not (numberp (numerator a)))
               (not (or (rzerop b)
                        (negativep (numerator b))))
               (fpp a)
               (fpp b)
               (rlessp a b))
          (and (rlessp (fp-mid a b) b)
               (rlessp a (fp-mid a b))))
 )


(defn find-func-zero
      (a b)
      (if (or (not (rlessp a b))
              (rlessp (mid-bound3) (rmagnitude a))
              (rlessp (mid-bound3) (rmagnitude b))
              (rlessp (fpmaximum)
                      (rquotient (fpminimum) (round-min)))
              (not (fpp a))
              (not (fpp b)))
          (rational 0 1)
          (if (or (and (numberp (numerator a))
                       (or (not (rlessp a
                                        (round (rtimes b (mid-bound1)))))
                           (not (rlessp (mid-bound2) b))))
                  (and (or (rzerop b)
                           (negativep (numerator b)))
                       (or (not (rlessp (round (rtimes a (mid-bound1)))
                                        b))
                           (not (rlessp (mid-bound2) (rneg a))))))
              (cons a b)
              (let ((mid (fp-mid a b)))
                   (if (equal (numberp (numerator (func a)))
                              (numberp (numerator (func mid))))
                       (find-func-zero mid b)
                       (find-func-zero a mid)))))
      ((lessp (find-func-zero-measure a b
                                      (fpminspace))))
     )

(prove-lemma rlessp-rmagnitude-x-x
           (rewrite)
           (not (rlessp (rmagnitude x) x))
           )
```

```
(prove-lemma fp-mid-bounded-fact
 (rewrite)
 (implies (and (rlessp a b)
               (not (rlessp (mid-bound3) (rmagnitude a)))
               (not (rlessp (mid-bound3) (rmagnitude b)))
               (not (rlessp (fpmaximum)
                            (rquotient (fpminimum) (round-min))))
               (fpp a)
               (fpp b)
               (or (not (numberp (numerator a)))
                   (and (rlessp a
                                (round (rtimes b (mid-bound1))))
                        (rlessp (mid-bound2) b)))
               (or (and (not (rzerop b))
                        (not (negativep (numerator b))))
                   (and (rlessp (round (rtimes a (mid-bound1)))
                                b)
                        (rlessp (mid-bound2) (rneg a)))))
          (not (rlessp (mid-bound3)
                       (rmagnitude (fp-mid a b)))))
 )


(prove-lemma find-func-zero-returns-close-values (rewrite)
             (implies
              (and
               (rlessp a b)
               (not (rlessp (mid-bound3) (rmagnitude a)))
               (not (rlessp (mid-bound3) (rmagnitude b)))
               (not (rlessp (fpmaximum)
                            (rquotient (fpminimum) (round-min))))
               (fpp a)
               (fpp b))
              (let ((lower (car (find-func-zero a b)))
                    (upper (cdr (find-func-zero a b))))
                (or (and (numberp (numerator lower))
                         (or (not (rlessp lower
                                          (round (rtimes upper (mid-bound1)))))
                             (not (rlessp (mid-bound2) upper))))
                    (and (or (rzerop upper)
                             (negativep (numerator upper)))
                         (or (not (rlessp (round (rtimes lower (mid-bound1)))
                                          upper))
                             (not (rlessp (mid-bound2) (rneg lower)))))))
             )

(prove-lemma find-func-zero-returns-a-zero (rewrite)
       (implies
         (and (not (equal (numberp (numerator (func a)))
                          (numberp (numerator (func b)))))
              (rlessp a b)
              (not (rlessp (mid-bound3) (rmagnitude a)))
              (not (rlessp (mid-bound3) (rmagnitude b)))
              (not (rlessp (fpmaximum)
                           (rquotient (fpminimum) (round-min))))
              (fpp a)
              (fpp b))
         (not (equal (numberp (numerator (func (car (find-func-zero a b)))))
                     (numberp (numerator (func (cdr (find-func-zero a b))))))))
 )
```

References

1. G. Barrett. "Formal Methods Applied to a Floating-Point Number System". *IEEE Transactions on Software Engineering 15* (May 1989), 611-621.

2. Bill Bevier. A Library for Hardware Verification. Internal Note 57, Computational Logic, Inc., June, 1988. Draft.

3. William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, William D. Young. "An Approach to Systems Verification". *Journal of Automated Reasoning 5* (November 1989).

4. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

5. Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional Instantiation in First Order Logic. Tech. Rept. 44, Computational Logic, Inc., Austin, Texas, May, 1989.

6. W. S. Brown. "A Simple but Realistic Model of Floating-Point Computation". *ACM Transactions on Mathematical Software 7*, 4 (December 1981).

7. William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions.* Prentice-Hall, New Jersey, 1980.

8. T.J. Dekker. Correctness Proof and Machine Arithmetic. In *Performance Evaluation of Numerical Software*, Fosdick, Eds., North-Holland, 1979.

9. Gries, D. *The Science of Computer Programming.* Springer-Verlag, 81.

10. John Erick Holm. *Floating-Point Arithmetic and Program Correctness Proofs*. Ph.D. Th., Cornell University, 1980.

11. IEEE Standards Board. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, 1988.

12. Matthew Kaufmann. A User’s Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Technical Report 19, Computational Logic, Inc., May, 1988.

13. D. E. Knuth. *The Art of Computer Programming. Volume 2/ Seminumerical Algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1969.

14. Richard Mansfield. "A Complete Axiomatization of Computer Arithmetic". *Mathematics of Computation 42*, 166 (April 1984).

15. Webb Miller. *The Engineering of Numerical Software.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

16. J Strother Moore. PITON: A Verified Assembly Level Language. Technical Report 22, Computational Logic, Inc., 1988.

17. B A Wichmann. "Towards a Formal Specification of Floating Point". *Computer Journal 32* (December 1989).

18. J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, 1963.

Table of Contents

1. Introduction 1
2. The Rational Library 4
2.1. Definitions 4
2.2. Rules 6
2.3. An example Use of R2 9
3. A Floating-Point System Axiomatization 11
3.1. The Axioms 11
3.2. A Model that Shows the Axioms to be Consistent 13
3.3. What is a Floating-Point Program? 13
4. An Example Floating-Point Program 15
4.1. **func** 15
4.2. **fp-mid** 15
4.3. **close** 17
4.4. **unreasonable** 18
4.5. Is **find-func-zero** a Floating-Point Program? 19
5. A Floating-Point Program Correctness Theorem 20
5.1. Correctness Argument Outline 20
5.2. An NQTHM Zero-Finding Program and Correctness Theorem 20
6. Future Work 23
6.1. Planned Work 23
6.2. Other Possible Work 23

Appendix A. An example R2 proof 25

Appendix B. The Rational Library 26

Appendix C. Floating-point Axiomatization 47

Appendix D. **fp-mid** and a Correctness Proof 57